

# 华中科技大学

## 课程实验报告

课程名称： 数据结构实验

专业班级 CS2110

学 号 U202115652

姓 名 李嘉鹏

指导教师 郑渤龙

报告日期 2022 年 6 月 1 日

计算机科学与技术学院

## 目 录

|                                     |            |
|-------------------------------------|------------|
| <b>1 基于顺序存储结构的线性表实现.....</b>        | <b>1</b>   |
| 1.1 问题描述 .....                      | 1          |
| 1.2 系统设计 .....                      | 2          |
| 1.3 系统实现 .....                      | 2          |
| 1.4 系统测试 .....                      | 10         |
| 1.5 实验小结 .....                      | 20         |
| <b>2 基于邻接表的图实现 .....</b>            | <b>21</b>  |
| 2.1 问题描述 .....                      | 21         |
| 2.2 系统设计 .....                      | 22         |
| 2.3 系统实现 .....                      | 23         |
| 2.4 系统测试 .....                      | 31         |
| 2.5 实验小结 .....                      | 41         |
| <b>3 课程的收获和建议 .....</b>             | <b>42</b>  |
| 3.1 基于顺序存储结构的线性表实现 .....            | 42         |
| 3.2 基于链式存储结构的线性表实现 .....            | 42         |
| 3.3 基于二叉链表的二叉树实现 .....              | 43         |
| 3.4 基于邻接表的图实现 .....                 | 43         |
| <b>参考文献 .....</b>                   | <b>44</b>  |
| <b>附录 A 基于顺序存储结构线性表实现的源程序 .....</b> | <b>45</b>  |
| <b>附录 B 基于链式存储结构线性表实现的源程序 .....</b> | <b>68</b>  |
| <b>附录 C 基于二叉链表二叉树实现的源程序 .....</b>   | <b>91</b>  |
| <b>附录 D 基于邻接表图实现的源程序 .....</b>      | <b>118</b> |

## 1 基于顺序存储结构的线性表实现

线性表是非常重要的数据结构之一，也是最常用的数据存储形式。其数据元素可以是数、符号或各类复杂信息等，因而在编写大型软件和程序时具有很好的应用价值。在本实验中，我以顺序表作为线性表的物理结构，通过完成线性表中常用的 18 个基本函数（包括 5 个附加功能），实现了对一个线性表的基本操作。进一步，我将全部函数的功能进行整合，并制作了一个可视化的、具有菜单选择画面的功能演示系统，从而实现了面对面的演示功能，加深了我对线性表的逻辑结构和物理结构的理解。

本实验的系统整体结构如下图1-1所示。

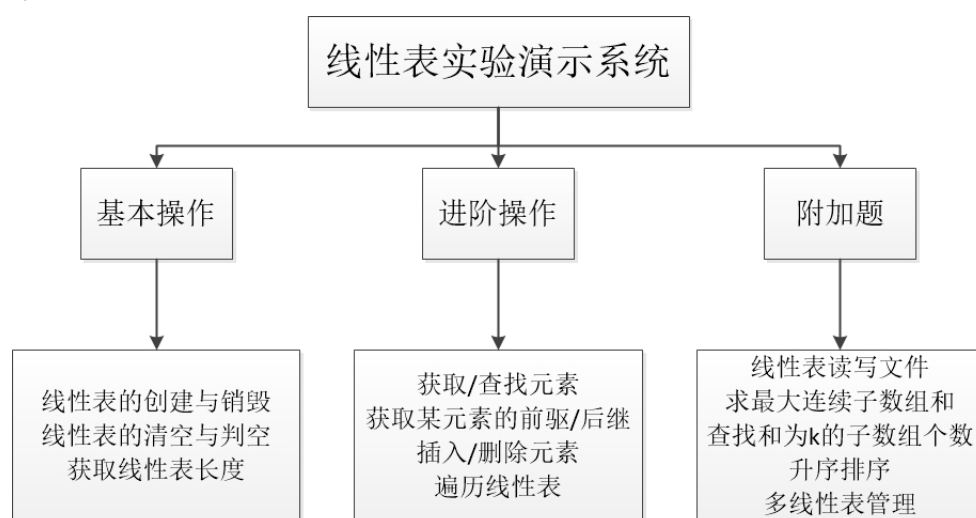


图 1-1 基于顺序存储结构的线性表系统结构

### 1.1 问题描述

根据华中科技大学 2021 级数据结构实验任务书<sup>[1]</sup>，实验一需要实现与线性表相关的 18 种功能。

其中，基本功能分别是：线性表的创建（初始化）、销毁、清空，判断线性表是否为空，求表的长度，获取（查找）线性表中某个元素，获取某个给定元素的前驱元素、后继元素，在给定位置向线性表中插入元素，遍历线性表。

附加功能包括：设计数据记录格式以实现线性表的文件形式保存（要求能保存线性表数据逻辑结构  $(D,R)$  的完整信息），统计线性表中和为  $k$  的连续子数组个数，对线性表中元素进行升序排序，以及多线性表管理（添加、查找和移除等

功能)。

## 1.2 系统设计

本实验演示系统采用菜单选择的方式，通过读取用户输入的功能编号执行对应的操作。实现选择的菜单代码如下1.1：

**算法 1.1.** 基于顺序存储结构的线性表系统菜单

```
while (op) {  
    system("cls");//清屏  
    printf(系统提示信息);  
    scanf("%d", &op);//读取用户输入  
    switch (op) {  
        // 以下分别对应 18 个功能，根据输入情况执行对应的操作  
        case 1:  
            InitList(L);  
        case 2:  
            DestroyList(L);  
        case 3:  
            ...  
    }  
}
```

使用时呈现出的菜单效果如图1-2所示。

## 1.3 系统实现

本系统由 18 个函数组成，下面分别说明各个函数的思想、算法与实现上的具体细节。（为便于解释，将线性表的名称设为 L。数组的编号由 0 开始）

### 1) 线性表的创建 InitList

功能：如果线性表 L 不存在，则调用此函数构造一个空的线性表并返回 OK，否则返回 INFEASIBLE



图 1-2 系统菜单

思路：为了判断线性表是否存在，需要考虑 `L.elem`（即表头指针）是否为空，若为空则使用 `malloc` 函数为线性表表头申请一块内存空间。需要注意的是，初始化时还需要将线性表的表长 `L.length` 赋为 0（因为是空表），其最大容量 `L.listsize` 应赋值为 `LIST_INIT_SIZE`。

## 2) 销毁线性表 `DestroyList`

功能：如果当前线性表存在，销毁线性表并释放数据元素空间，返回 `OK`，否则返回 `INFEASIBLE`

思路：这里判断一个线性表是否为空的方法与上面类似，若为空则对表头指针执行 `free` 操作并赋值为 `NULL`，由表头指针不存在就可以认为线性表已被销毁。

## 3) 清空线性表 `ClearList`

功能：如果线性表存在，则删除其中所有元素并返回 `OK`，否则返回 `INFEASIBLE`

思路：由于本实验中采用数组来存储数据元素，其内存由系统预先分配，无法进行 `free` 操作，因此只需将线性表表长置为 0。

## 4) 线性表判空 ListEmpty

功能：如果线性表存在，判断其是否为空，若为空则返回 TRUE，否则返回 FALSE；如果线性表不存在，返回 INFEASIBLE。

思路：首先，需要判断线性表的存在性，如果表头指针的值不等于 NULL 则说明线性表存在。接下来，还要进一步判断其中的元素个数，如果表长为 0 则说明线性表存在但为空。

## 5) 求线性表长度 ListLength

功能：如果线性表存在，返回其长度，否则返回 INFEASIBLE

思路：对于此后的每个函数，判断线性表是否存在的方法都与前述一致，后续不再赘述。若线性表不为空则直接返回其表长即可。

## 6) 获取元素（根据位置找元素）GetElem

功能：如果线性表存在，获取它的第  $i$  个元素并保存在  $e$  中，返回 OK；如果  $i$  不合法，返回 ERROR；如果线性表不存在，返回 INFEASIBLE

思路：首先判断所给  $i$  的合法性。如果  $i < 1$  或  $i > L.length$ ，显然输入有误；否则，将  $e$  赋值为  $L$  中的第  $i$  个元素即  $L.elem[i-1]$ 。

## 7) 查找元素（根据元素找位置）LocateElem

功能：如果线性表存在，由用户输入  $e$  的值，查找元素  $e$  在  $L$  中的位置并返回位置序号；如果  $L$  中不存在元素  $e$ ，返回 0；如果  $L$  不存在，返回 INFEASIBLE（即-1）

思路：首先判断线性表存在性。如果存在，则对线性表元素进行遍历，逐个比对线性表元素的值与  $e$  是否相等，如果找到则跳出遍历循环并返回当前位置，否则说明查找失败。本算法的时间复杂度为  $O(n)$ 。

## 8) 获取前驱元素 PriorElem

功能：如果线性表存在，获取  $L$  中元素  $e$  的前驱并保存在  $pre$  中，返回 OK；如果没有前驱，返回 ERROR；如果线性表不存在，返回 INFEASIBLE

思路：对存在的线性表元素进行遍历，如果找到了相应元素则记录其位置。

记录位置后还要继续比对，如果位置为 0 则说明该元素位于线性表的首位，不存在前驱元素。

## 9) 获取后继元素 NextElem

功能：如果线性表存在，获取 L 中元素 e 的后继并保存在 next 中，返回 OK；如果没有后继，返回 ERROR；如果线性表不存在，返回 INFEASIBLE

思路：类似于获取前驱元素的方法，同样遍历线性表并记录位置。如果得到的位置等于 L.length-1 则说明该元素位于线性表的末位，不存在后继元素。

## 10) 插入元素 ListInsert

功能：如果线性表存在，将元素 e 插入到 L 的第 i 个元素之前，返回 OK；当插入位置不正确时，返回 ERROR；如果线性表不存在，返回 INFEASIBLE

思路：首先检验 i 的合法性。如果  $i < 1$  或  $i > L.length + 1$  则输入不合法（函数要求插入第 i 个元素之前， $i < 1$  或  $i > L.length + 1$  时不存在这样的元素； $i = 1$  时，说明要在线性表的首位进行插入； $i = L.length + 1$  时，视为在线性表的末位进行插入）。插入前，需要注意当前线性表是否已满（即 L.length 是否已经达到 L.listsize），如果已达到存储空间的上限则需要使用 malloc 函数进行扩容，每次扩容都增加 LISTINCREMENT 的容量，如算法1.2所示。插入过程中，首先将从目标位置 i 开始的每个元素都向后移动一位，然后将原位置赋值为元素 e，L.length 增加 1，就完成了全部的插入操作。

### 算法 1.2. 判断是否达到容量上限并进行扩容

```
L.length++;  
if (L.length > L.listsize) {  
    ElemType *newbase = (ElemType *)realloc(L.elem, (L.listsize + LISTIN-  
        CREMENT) * sizeof(ElemType));  
    L.elem = newbase;  
    L.listsize += LISTINCREMENT;  
}
```

插入操作算法流程图如图1-3所示。

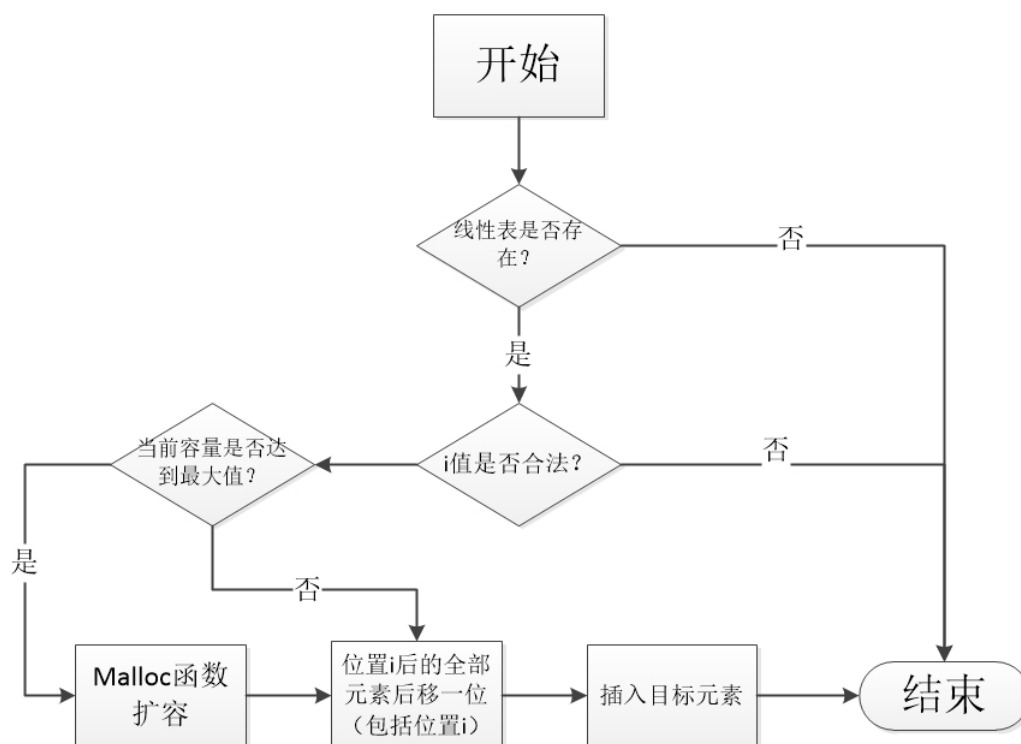


图 1-3 线性表的插入操作

## 11) 删除元素 ListDelete

功能：如果线性表存在，删除 L 的第 i 个元素并保存在 e 中，返回 OK；当删除位置不正确时，返回 ERROR；如果线性表不存在，返回 INFEASIBLE  
思路：首先同样判断 i 的合法性，若  $i < 1$  或  $i > L.length$  则输入有误，否则从给定位置开始，将后面的元素全部前移一位，并将 L.length 减 1。

## 12) 遍历线性表 ListTraverse

功能：如果线性表存在，依次显示其中的元素，每个元素间空一格，返回 OK；如果线性表不存在，返回 INFEASIBLE  
思路：直接对线性表进行遍历即可，每遍历一个元素就输出一次。

## 13) 线性表读写文件 SaveList/LoadList

功能：如果线性表存在，将 L 的元素写到 FileName 文件中并返回 OK，否则返回 INFEASIBLE。如果线性表不存在，将 FileName 文件中的元素写入 L 并返回 OK，否则返回 INFEASIBLE。



思路：读文件和写文件都采用文件指针完成。将线性表写入文件时，利用 `fprintf` 函数不断读入数据元素，直到读入了 `L.length` 个，如算法1.3所示。从文件中读取线性表时略有不同，首先要分配空线性表的内存空间，然后一个个读入数据，直到读取到文件的最后一个字符，如算法1.4所示。还要注意在读取的同时进行计数，以确定新表的表长。

## 算法 1.3. 写入文件

```
FILE *fp;
fp = fopen(FileName, "w");
for (int i = 0; i < L.length; i++)
    fprintf(fp, "%d ", L.elem[i]);
fclose(fp);
```

## 算法 1.4. 读取文件

```
FILE *fp;
fp = fopen(FileName, "r");
L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
// 创建新的数据空间
int len = 0;
while (!feof(fp))
    fscanf(fp, "%d", &L.elem[len++]);
// 从文件中写入数据到线性表中，直到文件结束
L.length = len - 1;
fclose(fp);
```

## 14) 统计和为 $k$ 的连续子数组个数 `SubArrayNum`

功能：在线性表中搜索和为  $k$  的连续子数组，成功返回其个数，否则返回 `ERROR`

思路：由于是连续子数组，所以其含有的元素个数只可能在 1 到 `L.length` 之间取值。按子数组中所含元素个数进行遍历，分别枚举出线性表所有可能存在的连续子数组并求和、比较，即可得出和为  $k$  的子数组总数。本算法总

共查找的次数为  $n+(n-1)+\dots+1=(n^2+n)/2$ ，时间复杂度为  $O(n^2)$ 。

## 15) 最大连续子数组和 MaxSubArray

功能：在线性表中求其连续子数组最大和，并返回这个值

思路：同样可以按子数组中所含元素个数对线性表进行遍历。由于要求最大值，则将初始变量  $\max$  置为 -999999（初始值足够小，确保任何一个连续子数组的和都大于它），在每轮遍历中都检查一次当前子数组的和是否大于  $\max$ ，如果是，则将  $\max$  的值替换为当前子数组的和，最后输出  $\max$  即可。本算法的时间复杂度为  $O(n^2)$ 。

算法流程图如图1-4所示。

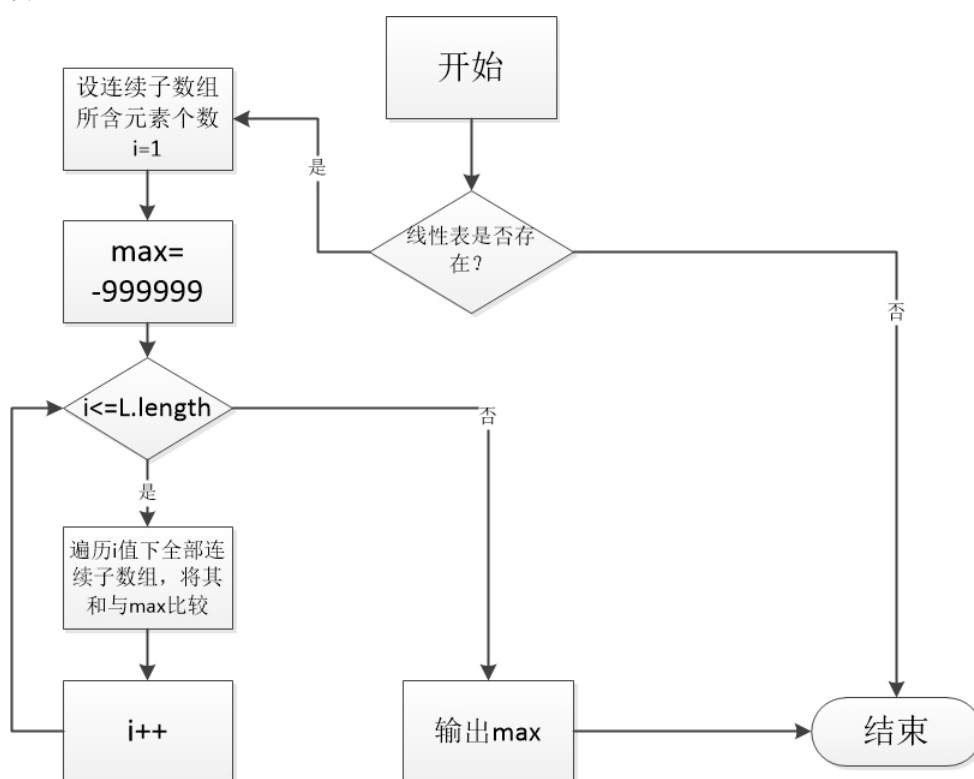


图 1-4 求最大连续子数组和

## 16) 对元素进行升序排序 SortList

功能：对线性表  $L$  进行升序排序，成功返回 OK，否则返回 ERROR

思路：此函数本质上就是对给定元素进行排序，我采用了冒泡排序（如果对相邻的两个元素，前一元素比后一元素大，则交换二者位置）的方式，冒泡  $L.length-1$  轮即可完成。本算法的时间复杂度为  $O(n^2)$ 。

## 17) 多线性表管理 MultiList

功能：设计数据结构，管理多个线性表的添加、移除、查找等

思路：由于要对多个线性表进行管理，所以要考虑设计一种包含多个线性表的线性表集合，定义如下：

```
typedef struct {
    struct {
        char name[30];
        SqList L;
    } elem[10];
    int length;
} LIST; //线性表集合结构定义
```

直观上看，该结构的示意图如图1-5所示。在 LIST 集合中，elem 数组最

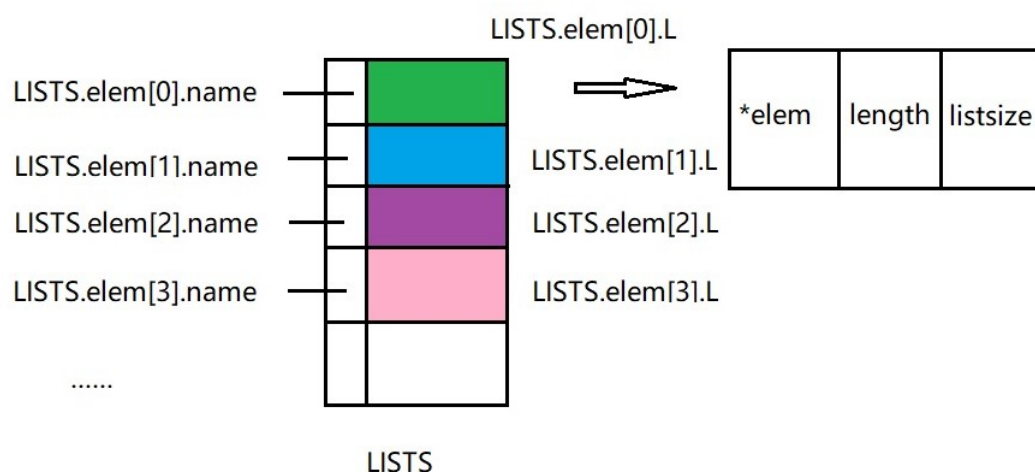


图 1-5 线性表集合结构

多可存放 10 个线性表，每个线性表包含名称信息和线性表结构（表头指针、表长、表的最大容量），LISTS.length 表示共存放了多少个线性表。对于添加操作，使用 AddList 函数在 Lists 中增加一个名称为 ListName 的空线性表，并使线性表集合长度增加 1；对于删除操作，首先判断线性表集合是否为空，若不为空则一一比较其中线性表的名称与所给名称是否相同，若相同则删除该线性表并使其后的线性表在 LISTS 中全部前移一位；查找操作的方法与删除操作中比较的方法一致。

## 1.4 系统测试

下面是根据不同测试输入得到的结果：

### 1) 线性表的创建

首次进入系统，输入 1，显示“线性表创建成功”，如图1-6所示。

图 1-6 线性表的创建（表不存在）

如果保留当前线性表，再次输入 1，显示“线性表创建失败”，如图1-7所示。

图 1-7 线性表的创建（表已存在）

### 2) 销毁线性表

如果已存在线性表，输入 2，显示“数据元素空间释放完成，线性表销毁成功”，如图1-8所示。

图 1-8 销毁线性表（表已存在）

如果线性表已经被销毁或没有创建线性表，显示“线性表销毁失败”，如图1-9所示。

图 1-9 销毁线性表（表不存在）

### 3) 清空线性表

如果已存在线性表，输入 3，显示“线性表中所有数据元素已删除，线性表

清空成功”，如图1-10所示。



图 1-10 清空线性表（表已存在）

否则显示“线性表清空失败”，如图1-11所示。



图 1-11 清空线性表（表不存在）

#### 4) 线性表判空

首次进入系统，输入 4，显示“线性表不存在”，如图1-12所示。




图 1-12 线性表判空（表不存在）

然后创建一个空线性表，输入 4，显示“线性表为空”，如图1-13所示。




图 1-13 线性表判空（表存在但为空）

向线性表中插入元素“1 2 -4 3”，显示“线性表不为空”，如图1-14所示。（说明：输入元素的方式是从文件直接读入，后续不再赘述，如图1-15所示）

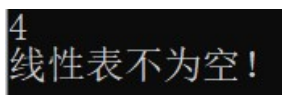


图 1-14 线性表判空（表存在且不为空）

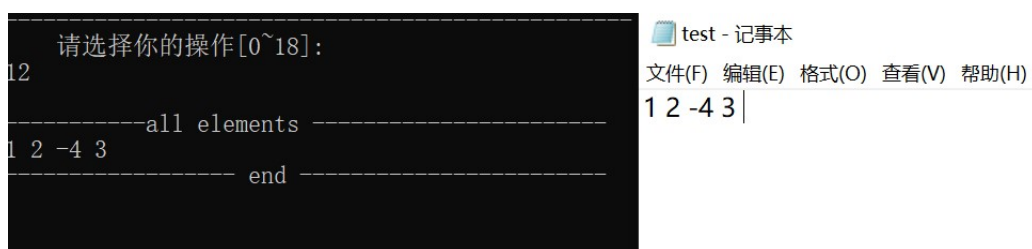


图 1-15 文件读入测试集示例

## 5) 求线性表长度

首次进入系统，输入 5，显示“线性表不存在”，如图1-16所示。

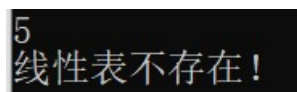


图 1-16 求线性表长度（表不存在）

创建一个空线性表，输入 5，显示“线性表的长度为 0”，如图1-17所示。



图 1-17 求线性表长度（表存在但为空）

向线性表中依次插入元素“1 2 -4 3”，显示“线性表的长度为 4”，如图1-18所示。



图 1-18 求线性表长度（正常情况）

## 6) 获取元素（根据位置找元素）

创建新线性表，输入“1 2 -4 3”，首先查询位置为 3 的元素，显示“线性表的第 3 个元素为-4”，如图1-19所示。

```
6
请输入要查询的元素位置序号: 3
线性表的第3个元素为-4!
```

图 1-19 获取元素（正常情况）

然后，分别查询位置为 0 和 6 的元素，均显示“所给 i 值不合法”，如图1-20、图1-21所示。

```
6
请输入要查询的元素位置序号: 0
所给i值不合法!
```

图 1-20 获取元素（i=0 的情况）

```
6
请输入要查询的元素位置序号: 6
所给i值不合法!
```

图 1-21 获取元素（i 超出表长的情况）

## 7) 查找元素（根据元素找位置）

创建新线性表，输入“1 2 -4 3”，首先查找值为 3 的元素的位置，显示“元素 3 在线性表中的位置序号为 4”，如图1-22所示。

```
7
请输入要查询的元素的值: 3
元素3在线性表中的位置序号为4!
```

图 1-22 查找元素（正常情况）

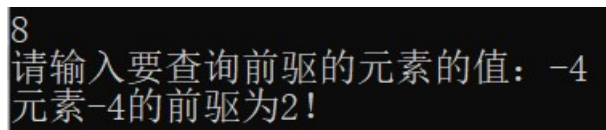
然后查询值为 100 的元素，显示“所给元素不存在”，如图1-23所示。

```
7
请输入要查询的元素的值: 100
所给元素不存在!
```

图 1-23 查找元素（元素不存在的情况）

## 8) 获取前驱元素

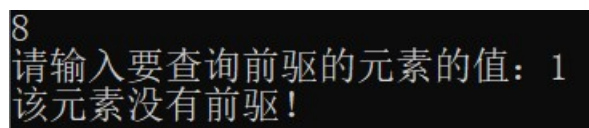
创建新线性表，输入“1 2 -4 3”，首先获取-4 的前驱元素，显示“元素-4 的前驱为 2”，如图1-24所示。



```
8
请输入要查询前驱的元素的值: -4
元素-4的前驱为2!
```

图 1-24 获取前驱元素（正常情况）

然后查询 1 的前驱元素，显示“该元素没有前驱”，如图1-25所示。

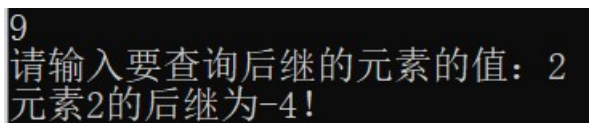


```
8
请输入要查询前驱的元素的值: 1
该元素没有前驱!
```

图 1-25 获取前驱元素（元素位于首位的情况）

## 9) 获取后继元素

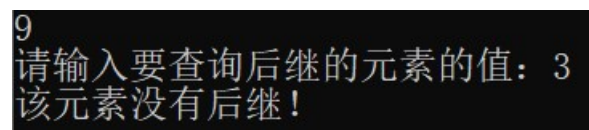
创建新线性表，输入“1 2 -4 3”，首先获取 2 的后继元素，显示“元素 2 的后继为-4”，如图1-26所示。



```
9
请输入要查询后继的元素的值: 2
元素2的后继为-4!
```

图 1-26 获取后继元素（正常情况）

然后查询 3 的前驱元素，显示“该元素没有后继”，如图1-27所示。



```
9
请输入要查询后继的元素的值: 3
该元素没有后继!
```

图 1-27 获取后继元素（元素位于末位的情况）

## 10) 插入元素

创建新线性表（此时表中无元素），插入元素 100（输入“1 100”），显示“插入成功”，遍历结果为“100”，如图1-28和图1-29所示。



```
10
请输入插入位置和插入元素的值，用空格隔开：1 100
插入成功！
```

图 1-28 插入元素（空表插入）

```
-----all elements -----
100
----- end -----
```

图 1-29 遍历结果 1

销毁该线性表，插入元素“1 2 -4 3”，输入“0 6”（意为在第 0 个元素前插入元素 6，所给位置不合法），显示“插入位置不正确”，如图1-30所示。

```
10
请输入插入位置和插入元素的值，用空格隔开：0 6
插入位置不正确！
```

图 1-30 插入元素（位置不合法）

输入 3 6，显示“插入成功”，遍历结果为“1 2 6 -4 3”，如图1-31所示。

```
-----all elements -----
1 2 6 -4 3
----- end -----
```

图 1-31 遍历结果 2

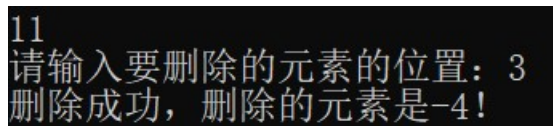
输入“10 7”（意为在第 10 个元素前插入元素 7，而此时表长为 5，所给位置不合法），显示“插入位置不正确”，如图1-32所示。

```
10
请输入插入位置和插入元素的值，用空格隔开：10 7
插入位置不正确！
```

图 1-32 插入元素（位置不合法）

## 11) 删除元素

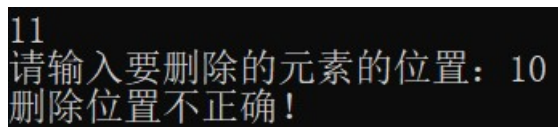
创建新线性表，输入“1 2 -4 3”，首先输入“3”删除位于第 3 位的元素，显示“删除成功，删除的元素是-4”，如图1-33所示。



```
11
请输入要删除的元素的位置: 3
删除成功，删除的元素是-4!
```

图 1-33 删除元素（正常情况）

输入“10”，显示“删除位置不正确”，如图1-34所示。



```
11
请输入要删除的元素的位置: 10
删除位置不正确!
```

图 1-34 删除元素（位置不合法）

## 12) 遍历线性表

首次进入系统，调用遍历函数，显示“线性表不存在”，如图1-35所示。



```
12
线性表不存在!
```

图 1-35 遍历线性表（表不存在）

创建新线性表（此时为空表），调用遍历函数，显示“线性表为空”，如图1-36所示。



```
12
线性表为空!
```

图 1-36 遍历线性表（表为空）

创建新线性表，输入“1 2 -4 3 8 7 11 -5 6”，调用遍历函数所得结果与输入相符，如图1-37所示。

```
12
-----all elements -----
1 2 -4 3 8 7 11 -5 6
-----end -----
```

图 1-37 遍历线性表（正常情况）

## 13) 线性表读写文件

读取文件：在程序目录下新建一个名为“test.txt”的文本文件，内容为“1 2 -4 3”。调用此函数并输入“test.txt”，显示“文件 test.txt 中的数据已成功读入到线性表中”，如图1-38所示。对该线性表进行遍历操作，结果与文本文件中的元素一致，如图1-39所示。

```
13
请输入目标文件名: test.txt
文件test.txt中的数据已成功读入到线性表中！
```

图 1-38 读文件

```
-----all elements -----
1 2 -4 3
-----end -----
```

图 1-39 读文件-遍历

然后销毁线性表，依次插入元素 10 20 30 40，调用此函数并输入“result.txt”，显示“线性表中的元素已成功写入文件 result.txt”，如图1-40所示。返回程序目录，打开此文本文件，其中内容与这四个元素一致，如图1-41所示。

```
13
请输入目标文件名: result.txt
线性表中的元素已成功写入文件result.txt！
```

图 1-40 写文件

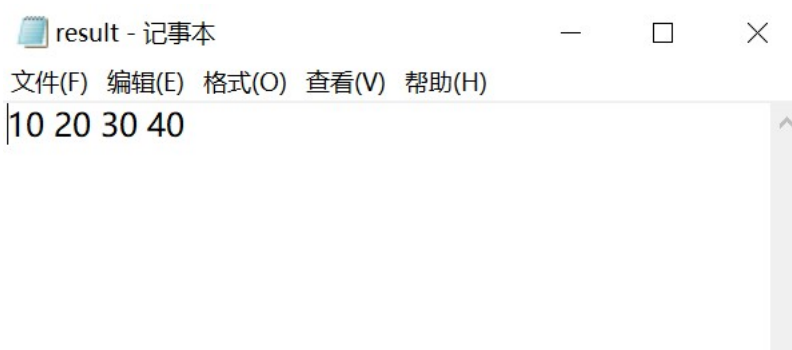


图 1-41 写文件-结果

## 14) 最大连续子数组和

创建新线性表，依次插入元素 1 2 -4 3，调用此函数，显示“线性表中连续子数组的最大和是 3”，如图1-42所示。

```
14
线性表中连续子数组的最大和是3
```

图 1-42 查找最大连续子数组和

## 15) 统计和为 k 的连续子数组个数

创建新线性表，依次插入元素 1 2 1 2 4 -1 3，调用此函数，输入“3”（代表要查找和为 3 的连续子数组个数），显示“线性表中和为 3 的连续子数组的个数是 5”，如图1-43所示。对此进行检验，和为 3 的连续子数组有（按从前到后的顺序）：[1,2]，[2,1]，[1,2]，[4,-1]，[3]，可见结果正确。

```
15
请输入想要查找连续子数组的和：3
线性表中和为3的连续子数组的个数是5
```

图 1-43 统计和为 3 的连续子数组个数

## 16) 对元素进行升序排序

创建新线性表，依次插入元素 5 4 2 6 1 -3，调用此函数，显示“线性表由小到大排序完成”，如图1-44所示。

```
16
线性表由小到大排序完成！
```

图 1-44 升序排序

然后对此线性表进行遍历操作，结果为“-3 1 2 4 5 6”，满足从小到大的条件，如图1-45所示。

```
-----all elements -----  
-3 1 2 4 5 6  
----- end -----
```

图 1-45 升序排序-遍历

## 17) 多线性表管理

调用此函数，显示“请选择功能：1. 线性表添加；2. 线性表移除；3. 线性表查找”，分别对应多线性表管理的三种功能。若输入“1”，系统提示“请输入线性表名称”，然后输入“LA”，系统提示“请依次输入该线性表的元素”，然后输入“1 2 -4 3”，显示“名为 LA 的线性表已加入线性表集合中”，如图1-46所示。

```
18  
请选择功能：1. 线性表添加；2. 线性表移除；3. 线性表查找  
1  
请输入线性表名称：LA  
请依次输入该线性表的元素，用空格隔开：1 2 -4 3  
名为LA的线性表已加入线性表集合中_
```

图 1-46 多线性表-添加

若输入“2”，系统提示“请输入线性表名称”，然后输入“LB”，显示“名为 LB 的线性表不存在”，如图1-47所示。

```
18  
请选择功能：1. 线性表添加；2. 线性表移除；3. 线性表查找  
2  
请输入线性表名称：LB  
名为LB的线性表不存在_
```

图 1-47 多线性表-删除

然后再以同样的方式增加一个新线性表 LC，调用此函数，输入“3”，系统提示“请输入线性表名称”，然后输入“LC”，显示“该线性表的位置是 2”，如图1-48所示。

```
18  
请选择功能：1. 线性表添加；2. 线性表移除；3. 线性表查找  
3  
请输入线性表名称：LC  
该线性表的位置是2
```

图 1-48 多线性表-查找

## 1.5 实验小结

通过本实验，我掌握了线性表的基本操作及其实现方式。同时，在编写程序的过程中，我也收获了很多实用的技巧和思想。由于要完成一个整体的系统，所以我在设计函数时特别注重函数之间的联系和通用性。当然，在调试中我也曾遇到不少问题，例如线性表表头指针初始未置为 NULL 而导致创建线性表失败（通过主函数中加入 `L.elem=NULL` 语句即可解决）。又如插入元素的函数中，一旦数据元素达到存储上限则需要扩充容量，否则会导致插入失败。在编写程序的前期阶段我始终未能注意这一点，从而导致在其上花费了较长时间，不过最终还是圆满完成。

通过不断优化程序，我逐一排查了各个部分可能存在的问题，最终得到了功能完善的线性表实验演示系统。在后续的程序设计中，我也会记住本次实验为我提供的宝贵经验，持续提高代码编写和排错能力。

## 2 基于邻接表的图实现

图（无向图、有向图等）也是一种意义重大的数据结构，如果需要表示顶点间多对多的关系（例如多个朋友之间的关系、不同城市间的距离与连通性等实际问题），就需要使用图来完成一系列功能。在本实验中，我通过邻接表的形式来实现图相关信息的存储，并基于此完成对无向图的 17 个函数（包括 5 个附加功能），实现了对无向图的一系列操作。通过本实验，我加深了对图的概念以及相关运算的理解，进一步掌握了图的逻辑结构与物理结构的关系与联系。

本实验的系统整体结构如下图2-1所示。

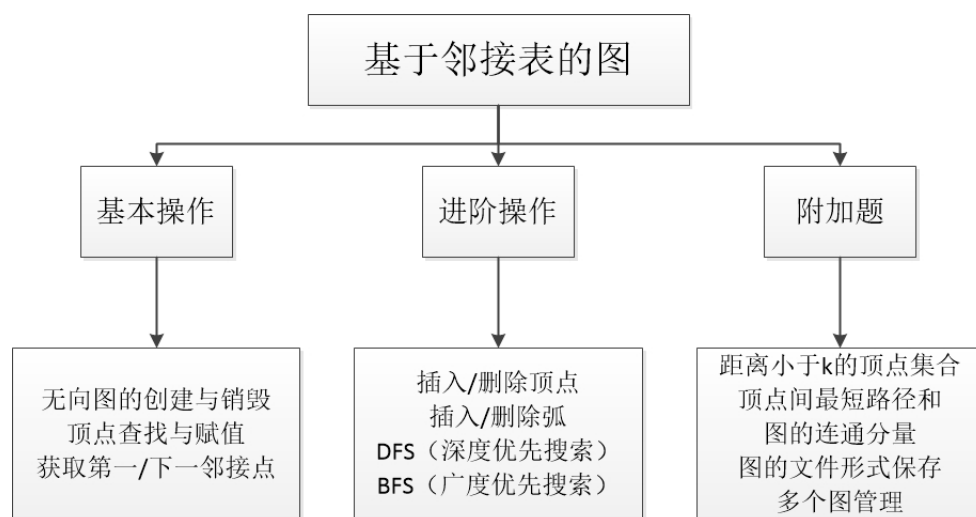


图 2-1 基于邻接表的图演示系统结构

### 2.1 问题描述

根据华中科技大学 2021 级数据结构实验任务书<sup>[1]</sup>，实验四需要基于邻接表实现与无向图相关的 18 种功能。

其中，基本功能分别是：无向图的创建与销毁，顶点的查找、赋值、插入与删除，获得第一邻接点、下一邻接点，弧的插入与删除，深度优先搜索遍历（DFS）和广度优先搜索遍历（BFS）。

附加功能包括：返回与某顶点距离小于  $k$  的顶点集合，求两个顶点之间最短路径的长度，求图的连通分量个数，实现图的文件形式保存，以及多个图管理（添加、查找和移除等功能）。



## 2.2 系统设计

本实验演示系统采用菜单选择的方式，通过读取用户输入的功能编号执行对应的操作。实现选择的菜单代码如下2.1：

### 算法 2.1. 基于邻接表的图演示系统菜单

```
while (op) {  
    system("cls");//清屏  
    printf(系统提示信息);  
    scanf("%d", &op);//读取用户输入  
    switch (op) {  
        // 以下分别对应 17 个功能，根据输入情况执行对应的操作  
        case 1:  
            CreateGraph(G);  
        case 2:  
            DestroyGraph(G);  
        case 3:  
            ...  
    }  
}
```

使用时呈现出的菜单效果如图2-2所示。





图 2-2 系统菜单

## 2.3 系统实现

本系统由 17 个函数组成，下面分别说明各个函数的思想、算法与实现上的具体细节。（为便于解释，将无向图的名称设为  $G$ ；本系统中每个函数都会检验图  $G$  是否为空，以下说明中默认图存在且不为空）

### 1) 创建无向图 CreateGraph

功能：根据图的顶点集  $V$  和图的关系集  $VR$  构造图  $G$  并返回 OK，如果  $V$  和  $VR$  不正确或有相同的关键字，返回 ERROR

思路：由于整个图要求每个顶点的关键字具有唯一性，因此首先要判断顶点关键字序列是否存在重复的情况，为此需要采用遍历操作并使用额外的一个数组来记录某个关键字是否出现。同时，还要检验输入的边的信息是否正确（两个端点是否存在、判断重复的边、排除自身与自身连接的情况等等）。最后创建邻接表，根据相应的信息生成结点，并给图的顶点数和边数（ $G.vexnum$  和  $G.arcnum$ ）赋初值。

判断关键字是否唯一和判断边的信息是否正确的代码如算法2.2和2.3:

## 算法 2.2. 判断关键字是否唯一

```
int check[100] = {0};
for (int i = 0; V[i].key != -1; i++) {
    if (check[V[i].key] == 1)
        return ERROR; // 如果关键字重复则输入有误
    check[V[i].key] = 1; // 如果关键字第一次出现, 则 check 赋值为 1
}
```

## 算法 2.3. 判断边的信息是否正确, 同时记录边数

```
int count = 0;
for (int i = 0; VR[i][0] != -1; i++) {
    if (check[VR[i][0]] != 1 || check[VR[i][1]] != 1)
        return ERROR; // 如果有一个顶点不存在则输入有误
    count++; // count 代表总共有几条边的信息
}
```

## 2) 销毁无向图 DestroyGraph

功能: 销毁无向图  $G$ , 删除  $G$  的全部顶点和边并返回 OK

思路: 由于无向图的邻接表是链表结构, 因此需要使用两个指针  $*temp$  和  $*pre$ , 其中  $temp$  每次循环中都首先向后移一位,  $pre$  始终保持在  $temp$  的前一位执行 free 操作。销毁结束后, 还要记得将图的顶点数和边数都置为 0。

## 3) 查找顶点 LocateVex

功能: 根据关键字  $u$  在图中查找顶点, 查找成功返回位序, 否则返回 -1

思路: 直接遍历  $G$  的全部顶点, 逐一比对关键字是否相同即可。本算法的时间复杂度为  $O(n)$ 。

## 4) 顶点赋值 PutVex

功能: 根据关键字  $u$  在图中查找顶点, 查找成功将该顶点值修改成  $value$  并

返回 OK；如果查找失败或关键字不唯一，返回 ERROR

思路：首先查找顶点，原理同上。为了确保赋值后顶点的关键字仍具有唯一性，还要对除该顶点以外的顶点的关键字进行遍历，逐一比对新关键字是否会产生冲突。如果没有问题则直接赋值即可。

## 5) 获得第一邻接点 FirstAdjVex

功能：根据关键字  $u$  在  $G$  中查找顶点，查找成功返回  $u$  的第一邻接顶点位序，否则返回 -1

思路：要查找顶点的第一个邻接顶点，则只需看该顶点对应的单链表是否存在第一个结点，存在则直接返回邻接点域 (adjvex) 的值即可。

## 6) 获得下一邻接点 NextAdjVex

功能：根据关键字  $v$  在图中查找顶点，查找成功返回  $v$  的邻接顶点相对于  $w$  的下一邻接顶点的位序，查找失败返回 -1

思路：首先分别判断  $v$  和  $w$  对应的顶点是否都存在。如果存在，则查找  $w$  在  $v$  的邻接表中的位序，并返回其下一个结点。

## 7) 插入顶点 InsertVex

功能：在图中插入顶点  $v$ ，成功返回 OK，否则返回 ERROR

思路：首先还是判断新顶点的关键字是否与已存在的顶点关键字产生重复，如果输入数据无误则直接在邻接表中插入。

## 8) 删除顶点 DeleteVex

功能：在图中删除关键字  $v$  对应的顶点以及相关的弧，成功返回 OK，否则返回 ERROR

思路：首先判断图是否为空；如果不为空，则顶点数应该大于 1（因为要保证删除节点后图不空）。接下来查找目标顶点的位序，并将该顶点后的全部顶点都前移一位。对于被删除的顶点，遍历统计其邻接点数量，这就是无向图在删除操作中弧的数量减少的数量。然后删除相关的弧，对剩余的全部顶点进行遍历并用 -999 标记要删除的弧结点。删除完后还要注意：由于该顶点后的全部顶点的位序都减了 1，所以现在仍然存在的弧中，有一部分弧

（它们的两个端点至少有一个是这些位序减了 1 的顶点）的位序信息也要做相应的调整。最后还要记得将无向图的顶点数减去 1。

算法流程图如图2-3所示。

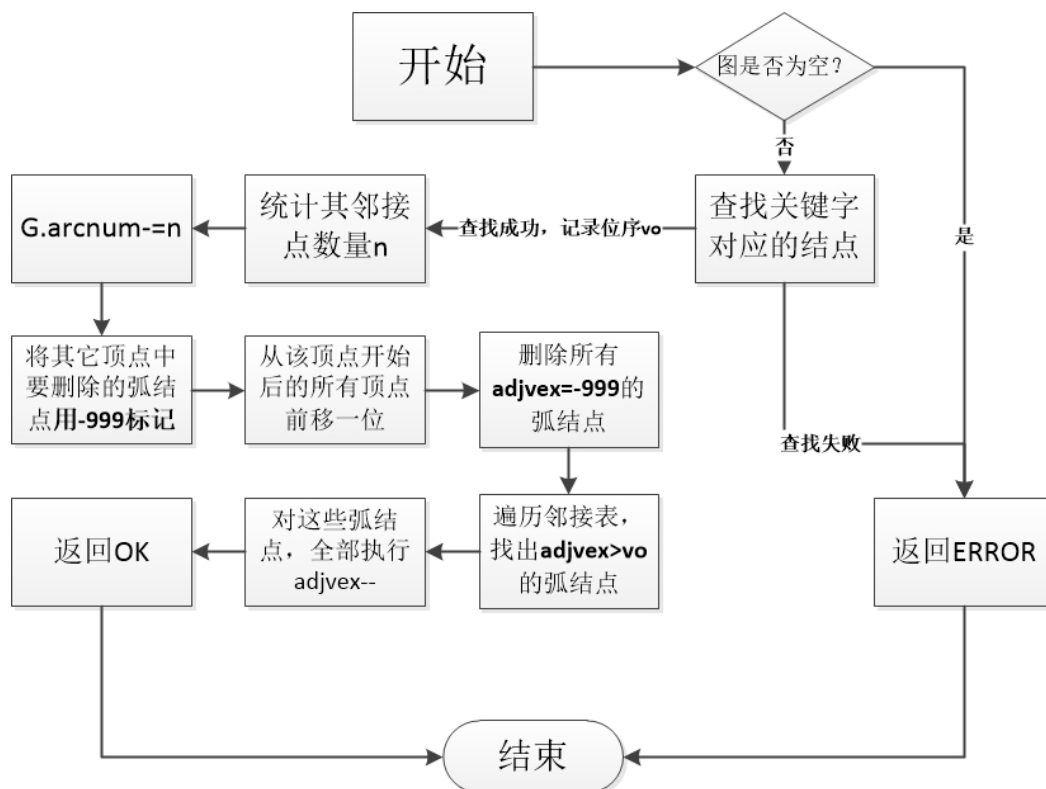


图 2-3 删除顶点

遍历查找待删除弧结点的代码如算法2.4。

## 算法 2.4. 遍历查找待删除的弧结点

```

ArcNode *temp;
temp = G.vertices[j].firstarc;
while (temp) {
    if (temp->adjvex == i)
        temp->adjvex = -999; //用-999 标记要删的弧结点
    else if (temp->adjvex > i)
        temp->adjvex--; //序号前移 1
    temp = temp->nextarc;
}
  
```

## 9) 插入弧 InsertArc

功能：在图中增加弧  $\langle v, w \rangle$ ，成功返回 OK，否则返回 ERROR

思路：首先判断两个顶点的存在性以及该路径是否已经存在。如果路径已存在，则不需做任何操作；否则还需要在两个顶点各自的邻接链表中分别创建一个新结点，来存储新增加的边的信息。还要记得将 G.arcnum 加 1。

## 10) 删除弧 DeleteArc

功能：在图 G 中删除弧  $\langle v, w \rangle$ ，成功返回 OK, 否则返回 ERROR

思路：首先判断两个顶点的存在性以及该路径是否已经存在。如果都存在，那么删除这两个弧结点即可。还要记得将 G.arcnum 减去 1。

## 11) 深度优先搜索遍历 DFSTraverse

功能：对图进行深度优先搜索遍历，依次对图中每一个顶点使用函数 visit 访问一次，且仅访问一次

思路：在遍历过程中，为了判断哪些顶点已被遍历过，考虑引入一个 visited 数组来记录某点是否已被访问。然后采用非递归的方式，按邻接表的顺序开始遍历，如果某顶点未被访问，则访问一次，然后访问它的第一邻接点，再访问第一邻接点的第一邻接点……这实质上是从邻接表中的第一个顶点开始一路访问下去，如果发现某顶点已被访问过，就从另外一个新的顶点开始重复上述操作。这样，图中所有的顶点最终都会被访问一次。算法流程图如图2-4所示。

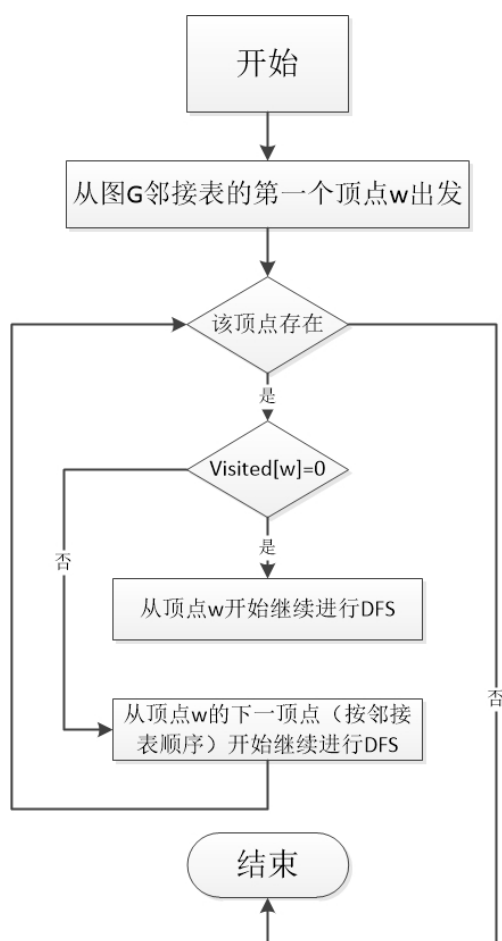


图 2-4 深度优先搜索

对一个图进行深度优先搜索，示例如图2-5，其中 1,2,...,7 为各顶点被访问的先后顺序。

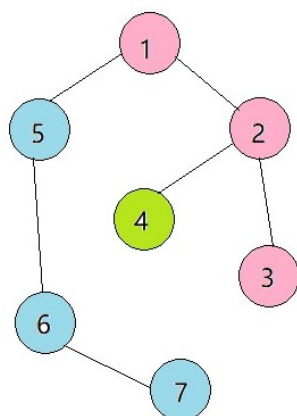


图 2-5 深度优先搜索-示例

## 12) 广度优先搜索遍历 BFSTraverse

功能：对图进行广度优先搜索遍历，依次对图中每一个顶点使用函数 `visit` 访问一次，且仅访问一次

思路：同样用 DFS 的方式记录顶点的访问情况。对于广度优先搜索，考虑使用队列的方式进行遍历。首先，将邻接表中的第一个顶点入队。此时，先访问队列中的唯一元素，在访问的同时将该元素出队，并将它的全部邻接点入队（第一轮）；然后继续访问队列中的元素，访问一次出队一个元素，并将此元素的全部邻接点入队（第二轮），一直持续下去。这里的主要想法是从第一个顶点开始，以“某顶点到第一个顶点的边数”作为访问的先后依据。对一个图进行广度优先搜索，示例如图2-6，其中 1,2,...,7 为各顶点被访问的先后顺序。

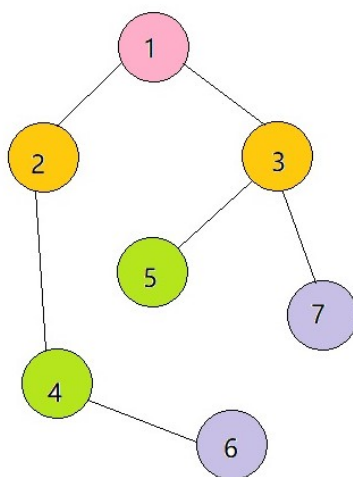


图 2-6 广度优先搜索-示例

## 13) 无向图读写文件 SaveGraph/LoadGraph

功能：如果无向图存在，将 `G` 的元素写到 `FileName` 文件中并返回 `OK`，否则返回 `INFEASIBLE`。如果无向图不存在，将 `FileName` 文件中的元素写入 `G` 并返回 `OK`，否则返回 `INFEASIBLE`。

思路：读文件和写文件都采用文件指针完成。对于写入文件，我设计了一种便于程序读取的数据存储方式，首先存储 `G` 的顶点数 `G.vexnum` 和边数 `G.arcnum`，接下来 `G.vexnum` 行分别存储每个顶点的邻接点数，再接下来 `G.vexnum` 行存储每个顶点的邻接点位序。这样，在从文件读取无向图信息的时候就大大降低了输入的难度。对于读取文件，方法也大体一致，通过调

用 `CreateGraph` 函数即可快速完成读入。

## 14) 返回距离小于 $k$ 的顶点集合 `VerticesSetLessThanK`

功能：返回与顶点  $v$  距离小于  $k$  的顶点集合

思路：首先要求出顶点  $v$  与其它全部顶点的最短路径，然后一一比较，如果距离小于  $k$  则满足条件。（具体的求法见下面函数）

## 15) 求两个顶点间的最短路径 `ShortestPathLength`

功能：返回顶点  $v$  与顶点  $w$  的最短路径长度

思路：在无向图中，求两个顶点的最短路径主要采用 Dijkstra 算法：从起始点开始，采用贪心算法的策略，每次遍历到起始点距离最近且未访问过的顶点的邻接点，直到达到终点。在本函数中，具体实现方式为：

首先声明一个数组  $d$  来保存起始点到各个顶点的最短距离，利用 `visited` 数组来记录已被访问的顶点，并声明一个变量。开始时顶点  $v$  的路径权重被赋为 0 ( $d[v]=0$ )。若对于顶点  $v$  存在一条边  $(v,t)$ ，则把  $d[t]$  设为 1，同时把所有其它（ $v$  不能直接到达的）顶点的路径长度设为无穷大 ( $INF=-999999$ )。然后从  $d$  数组中选择最小值（此函数中就是 1），该值就是顶点  $v$  到该值对应的顶点的最短路径，并把到该点的最短路径  $d[t]$  进行修改。但此时并不能保证这个顶点一定是最优的解，还需要考虑两点：新加入的顶点是否可以到达其它顶点；以及通过该顶点到达其它点的路径是否比从顶点  $v$  直接到达更短，如果是，那么就替换这些顶点在  $d$  中的值。然后，又一次从  $d$  中找出最小值，不断重复上述操作直到找到  $d[w]$ ，即顶点  $v$  与顶点  $w$  的最短路径。

## 16) 求图的连通分量个数 `ConnectedComponentsNums`

功能：返回无向图连通分量的个数

思路：此函数的原理与 DFS 中用到的队列思想类似，如果 DFS 中遍历到某点时队为空，则说明已经走完了一个完整的连通分量，因此只需要统计整个遍历过程中队为空的情况出现了几次，这就是图的连通分量的个数。

## 17) 多个图管理 `MultiGraph`

功能：管理多个无向图的添加、移除、查找等



思路：由于要对多个无向图进行管理，所以要考虑设计一种包含多个无向图的集合，定义如下：

```
typedef struct {  
    struct {  
        char name[30];  
        ALGraph G;  
    } elem[100];  
    int length;  
} GRAPHS; //无向图集合结构定义
```

在 GRAPHS 集合中，elem 数组最多可存放 100 个无向图，每个图包含名称信息和邻接表指针，GRAPHS.length 表示共存放了多少个无向图。对于添加操作，只需要调用创建无向图的函数 CreateGraph；对于删除操作，首先判断无向图集合是否为空，若不为空则一一比较其中图的名称与输入名称是否相同，若相同则删除该图并使其后的图在 GRAPHS 中全部前移一位；查找操作的方法与删除操作中比较的方法一致（如果查找成功则自动将当前系统的无向图图指针赋为已找到的无向图，便于进一步操作）。

## 2.4 系统测试

下面是根据不同测试输入得到的结果：

### 1) 创建无向图



图 2-7 创建无向图

## 2) 销毁无向图



图 2-8 销毁无向图

## 3) 查找顶点

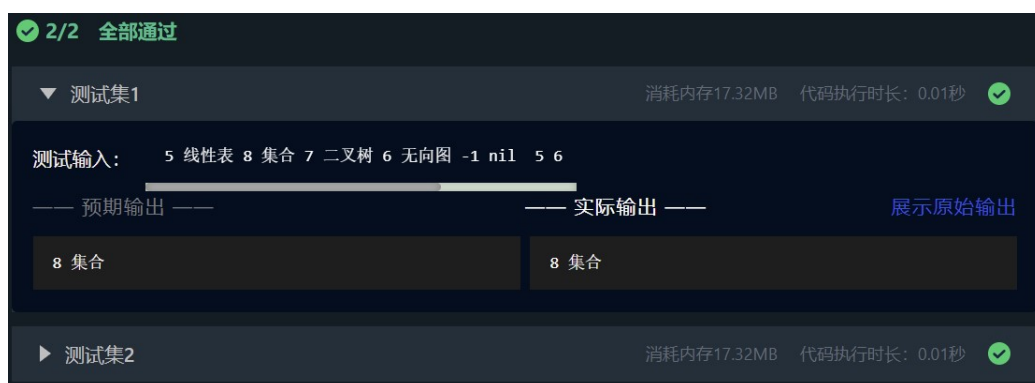


图 2-9 查找顶点

## 4) 顶点赋值



图 2-10 顶点赋值

## 5) 获得第一邻接点

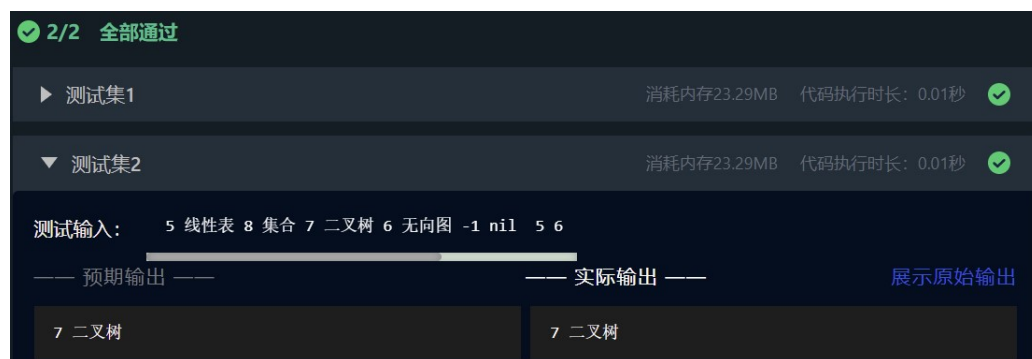


图 2-11 获得第一邻接点

## 6) 获得下一邻接点



图 2-12 获得下一邻接点

## 7) 插入顶点



图 2-13 插入顶点

## 8) 删除顶点

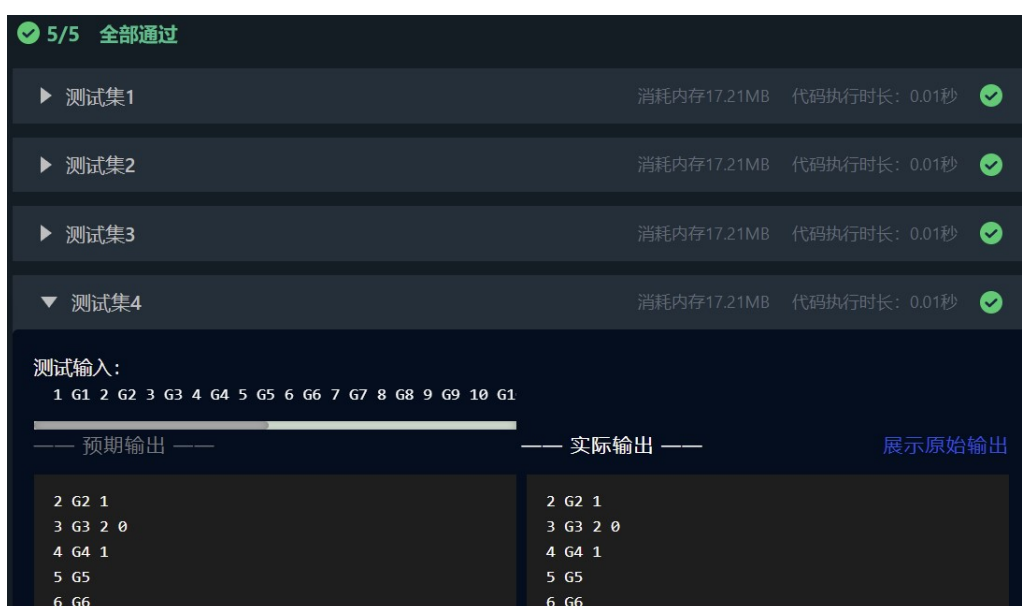


图 2-14 删除顶点

## 9) 插入弧



图 2-15 插入弧

## 10) 删除弧

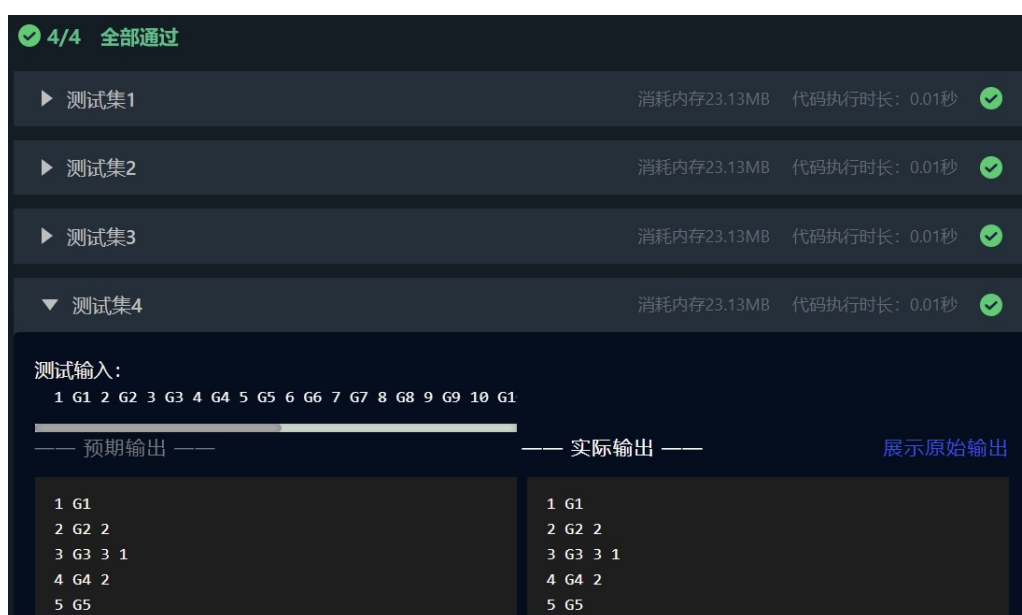


图 2-16 删除弧

## 11) 深度优先搜索遍历

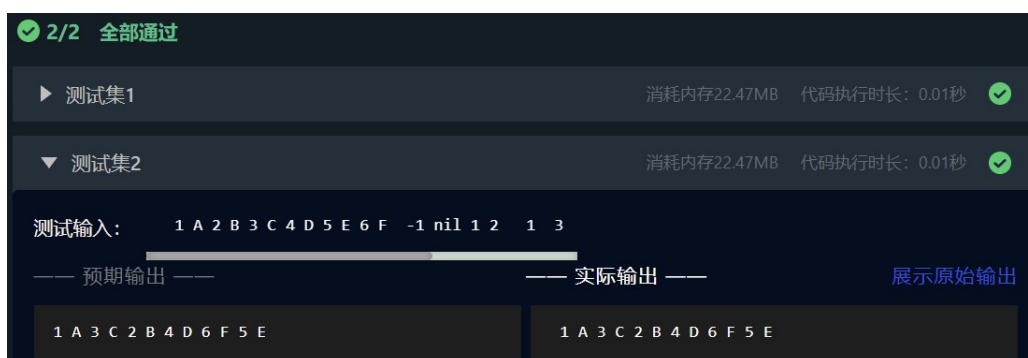


图 2-17 深度优先搜索遍历

## 12) 广度优先搜索遍历



图 2-18 广度优先搜索遍历

## 13) 无向图读写文件



图 2-19 无向图读写文件

## 14) 返回距离小于 k 的顶点集合

创建新无向图，输入“5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1”。输入“14”，系统提示“请输入顶点的关键字和 k 值（距离小于 k），用空格隔开”。然后输入“5 2”（意为寻找与顶点“5 线性表”距离小于 2 的顶点集合），显示“7 二叉树 6 无向图”，如图2-20所示。

```

14
请输入顶点的关键字和k值（距离小于k），用空格隔开：5 2
-----与顶点5距离小于2的顶点集合-----
7 二叉树 6 无向图
-----end-----
    
```

图 2-20 返回距离小于 k 的顶点集合

#### 15) 求两个顶点间的最短路径

创建新无向图，输入“5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1”，得到具有四个顶点的无向图。输入“15”，系统提示“请输入两个顶点的关键字，用空格隔开”。然后输入“5 8”（意为寻找顶点“5 线性表”和“8 集合”之间的最短路径），显示“顶点 5 和 8 之间最短路径的长度为 2”，如图2-21所示。

```

15
请输入两个顶点的关键字，用空格隔开：5 8
顶点5和8之间最短路径的长度为2!
    
```

图 2-21 求两个顶点间的最短路径（正常情况）

销毁此无向图，输入“1 G1 2 G2 3 G3 4 G4 5 G5 6 G6 7 G7 8 G8 9 G9 10 G10 11 G11 12 G12 13 G13 14 G14 15 G15 16 G16 17 G17 18 G18 19 G19 20 G20 -1 nil -1 -1”，输入“15”，然后输入“1 2”（意为寻找顶点“1 G1”和“2 G2”之间的最短路径），显示“输入数据错或这两个顶点之间不存在通路”，如图2-22所示。（显然这 20 个顶点之间相互独立，顶点 1 和 2 之间不存在通路）

```

15
请输入两个顶点的关键字，用空格隔开：1 2
输入数据错或这两个顶点之间不存在通路!
    
```

图 2-22 求两个顶点间的最短路径（通路不存在的情况）

#### 16) 求图的连通分量个数

创建新无向图，输入“5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1”，然后输入“16”，显示“无向图中连通分量的个数为 1”，如图2-23所



示。

```
16
无向图中连通分量的个数为1!
```

图 2-23 求图的连通分量个数（整个图为一个极大连通分量）

销毁此无向图，输入“1 G1 2 G2 3 G3 4 G4 5 G5 6 G6 7 G7 8 G8 9 G9 10 G10 11 G11 12 G12 13 G13 14 G14 15 G15 16 G16 17 G17 18 G18 19 G19 20 G20 -1 nil -1 -1”，然后输入“16”，显示“无向图中连通分量的个数为 20”，如图2-24所示。

```
16
无向图中连通分量的个数为20!
```

图 2-24 求图的连通分量个数（一个顶点为一个连通分量）

## 17) 多个图管理

调用此函数，显示“请选择功能：1. 无向图添加；2. 无向图移除；3. 无向图查找”，分别对应多个图管理的三种功能。若输入“1”，系统提示“请输入新无向图名称”，然后输入“新图”，系统提示“请依次输入该线性表的元素”，然后输入“5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1”，显示“无向图新图添加成功”，如图2-25所示。然后同样地加入上个函数所用的第二个图“20 个点”，如图2-26所示。

```
17
请选择功能：1. 无向图添加；2. 无向图移除；3. 无向图查找
1
请输入新无向图名称：新图
请输入顶点序列和关系对序列，分别以-1 nil和-1 -1结束：
5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1
无向图新图添加成功!
```

图 2-25 无向图添加-“新图”

```
17
请选择功能：1. 无向图添加；2. 无向图移除；3. 无向图查找
1
请输入新无向图名称：20个点
请输入顶点序列和关系对序列，分别以-1 nil和-1 -1结束：
1 G1 2 G2 3 G3 4 G4 5 G5 6 G6 7 G7 8 G8 9 G9 10 G10 11 G11 12 G1
2 13 G13 14 G14 15 G15 16 G16 17 G17 18 G18 19 G19 20 G20 -1 nil
-1 -1
无向图20个点添加成功！
```

图 2-26 无向图添加- “20 个点”

若输入“2”，系统提示“请输入无向图名称”，输入“GA”，显示“无向图集合中没有名为 GA 的无向图”，如图2-27所示。然后输入“20 个点”，显示“无向图 20 个点已从无向图集合中删除”，如图2-28所示。

```
17
请选择功能：1. 无向图添加；2. 无向图移除；3. 无向图查找
2
请输入无向图名称：GA
无向图集合中没有名为GA的无向图！
```

图 2-27 无向图删除（图不存在的情况）

```
17
请选择功能：1. 无向图添加；2. 无向图移除；3. 无向图查找
2
请输入无向图名称：20个点
无向图20个点已从无向图集合中删除！
```

图 2-28 无向图删除（正常情况）

若输入“3”，系统提示“请输入无向图名称”，输入“新图”，显示“查找成功”，如图2-29所示。此时系统的图邻接表指针已经被赋为“新图”对应的邻接表指针，进行遍历操作，结果无误，如图2-30所示。

```
17
请选择功能：1. 无向图添加；2. 无向图移除；3. 无向图查找
3
请输入无向图名称：新图
查找成功！
```

图 2-29 无向图查找（图不存在的情况）

```
11
-----深度优先搜索遍历结果-----
5 线性表 7 二叉树 8 集合 6 无向图
-----end-----
```

图 2-30 无向图查找（正常情况）

## 2.5 实验小结

“基于邻接表的图实现”是本学期数据结构实验课的最后实验，为我这学期的程序设计画上了一个完美的句号。虽然本实验看似只是采用了邻接表这一“专用于图”的数据结构，但它本质上是对前几次实验的一种综合应用（如顺序表和链表），在“基于二叉链表的二叉树实现”实验中频繁出现的递归函数及其思想也为我提供了许多帮助。

在调试过程中我也遇到了一些问题，例如：在使用双指针进行遍历的时候没有考虑到临界条件，可能会导致某个指针访问了不存在的地址（成为“野指针”）而导致程序崩溃。这需要我对特殊情况（例如遍历到最后或者某顶点不存在邻接点）进行特判，全面考虑到每一种可能性。还比如在删除弧的函数中，我简单地认为后台测试程序会检测弧数量的变化，所以没有加入对 `G.arcnum` 的修改语句，但实际上需要自己判断删除了多少条弧，导致输出时出现了问题。

通过本次实验，我更加熟练地掌握了图的基本运算，并进一步提高了自己编写代码的实践能力，受益匪浅。

## 3 课程的收获和建议

本学期的数据结构实验课对基于顺序存储的线性表（数组）、基于链式存储结构的线性表（单链表）、基于二叉链表的二叉树和基于邻接表的图四种常见的数据存储结构都有涉及。在这些实验中，我充分将课上所学的理论知识进行应用，依靠 C 语言代码完成了系统实现，巩固了我对其数据处理方式、物理结构及逻辑结构的理解。

暑假的数据结构课设将会对我的各方面能力提出更高更难的要求，但从这四次实验中，我深刻了解了不同功能函数的具体构建方法，以及面对问题应如何解决的思维模式。同时，还培养了我处理未知漏洞时谨慎耐心的态度。希望我能通过不断的练习充分打磨自身能力，更好地完成接下来的学习任务。

### 3.1 基于顺序存储结构的线性表实现

通过本实验，我加深了对线性表的概念和基本运算的理解，更加熟练地掌握了线性表的逻辑结构与物理结构的关系，同时熟练掌握了顺序表基本运算的实现。

由于采用的是顺序存储结构，即通过数组来存储数据元素，所以在具体实现上并不会过于困难，且比较便于对元素进行访问。实验中对元素的查找、替换、插入和删除等操作中蕴含的思想，为我后续的实验打下了基础，使我在完成单链表实验时更加顺利。

### 3.2 基于链式存储结构的线性表实现

单链表也是线性表的一种物理结构，由于其特殊的结构，在访问元素时并不像数组一样简单，但是它更能表现出元素之间的关系和联系，同时在移动元素时也比数组更加快捷。

通过本实验，我熟练掌握了单链表的性质与特点，既对上学期 C 语言课程中所学的链表内容进行了回顾，也在更大程度上发挥了它的功用。同时，在本实验中，由于对链表结点的访问时常需要通过指针来完成，且一些边界条件在初上手时比较容易被忽略，我还深刻体会到了野指针的危害，这有助于我在未来的程序设计中进一步规范代码严谨性。

## 3.3 基于二叉链表的二叉树实现

通过本实验，我加深了对二叉树的概念、基本运算以及二叉树的逻辑结构、物理结构关系的理解，熟练掌握了二叉树基本运算的实现。

本实验在单链表的基础上进行拓展，通过将二叉链表作为物理结构来表示二叉树的结构。经过实验二的练习，我在本实验中格外注意避免指针访问错误内存的问题，因而在编程过程中较为顺利。并且，由于二叉链表的根节点、根节点的左孩子和根节点的右孩子三者之间有着紧密的联系，所以在实现许多功能（例如判定二叉树的深度、进行先、中、后序遍历等）都要用到递归函数，这进一步丰富了我用递归思想解决实际问题的经验，也更加深刻地理解了递归的本质。

## 3.4 基于邻接表的图实现

通过本实验，我加深了对图的概念和基本运算的理解，熟练掌握了图的逻辑结构与物理结构的关系。虽然本实验重点在于无向图，但无向图和有向图、网具有很大相似之处。

本实验以邻接表作为物理结构，使我进一步了解了邻接表的结构、用法与特性。由于邻接表的定义中各类变量较为繁杂，例如顶点集采用数组存储而单个顶点的邻接点集合采用单链表存储，其中的关系也比较难以理清，所以这对我形成严谨审慎的编程风格有重要意义。并且，对顶点关键字和对弧的检验过程也培养了我细心分析、仔细思考的习惯。

图的应用与实际生活联系十分紧密，本实验也为我在处理生活中某些问题上提供了一种新的思路。

## 参考文献

- [1] 华中科技大学计算机科学与技术学院数据结构课程组. 2021 级数据结构实验任务书 [J], 2021.
- [2] 严蔚敏, 吴伟民. 数据结构 (C 语言版) [M]. [S.l.]: 清华大学出版社, 1997.
- [3] 严蔚敏, 吴伟民, 米宁. 数据结构题集 (C 语言版) [M]. [S.l.]: 清华大学出版社, 2007.
- [4] 殷立峰. Qt C++ 跨平台图形界面程序设计基础 [M]. [S.l.]: 清华大学出版社, 2014.
- [5] NYHOFF L. ADTs, Data Structures, and Problem Solving with C++[M]. 2005.

## 附录 A 基于顺序存储结构线性表实现的源程序

```
/* Linear Table On Sequence Structure */
```

```
#include <stdio.h>
```

```
#include <malloc.h>
```

```
#include <stdlib.h>
```

```
//各类常量定义如下
```

```
#define TRUE 1
```

```
#define FALSE 0
```

```
#define OK 1
```

```
#define ERROR 0
```

```
#define INFEASIBLE -1
```

```
#define OVERFLOW -2
```

```
#define LIST_INIT_SIZE 100
```

```
#define LISTINCREMENT 10
```

```
typedef int status; //数据元素种类定义
```

```
typedef int ElemType;
```

```
typedef struct {
```

```
ElemType *elem;
```

```
int length;
```

```
int listsize;
```

```
} SqList; //线性表结构定义
```

```
typedef struct {
```

```
struct {
```

```
char name[30];
```

```
SqList L;
```

```
} elem[10];
```

```
int length;
```

```
} LISTS; //线性表集合结构定义
```

```
LISTS Lists;
```

```
status InitList(SqList &L) {  
    // 线性表 L 不存在，构造一个空的线性表，返回 OK，否则返回 INFEASIBLE。  
    if (!L.elem) {  
        L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));  
        L.length = 0;  
        L.listsize = LIST_INIT_SIZE;  
        return OK;  
    } else  
        return INFEASIBLE;  
}
```

```
status DestroyList(SqList &L) {  
    // 如果线性表 L 存在，销毁线性表 L，释放数据元素的空间，返回 OK，否则返回 INFEASIBLE。  
    if (!L.elem)  
        return INFEASIBLE;  
    else {  
        free(L.elem);  
        L.elem = NULL;  
        return OK;  
    }  
}
```

```
status ClearList(SqList &L) {  
    // 如果线性表 L 存在，删除线性表 L 中的所有元素，返回 OK，否则返回 INFEASIBLE。  
    if (!L.elem)
```



```
return INFEASIBLE;
else {
    L.length = 0;
    return OK;
}
}
```

```
status ListEmpty(SqList L) {
    // 如果线性表 L 存在, 判断线性表 L 是否为空, 空就返回 TRUE, 否则返回 FALSE;
    // 如果线性表 L 不存在, 返回 INFEASIBLE。
    if (!L.elem)
        return INFEASIBLE;
    else {
        if (L.length == 0)
            return TRUE;
        else
            return FALSE;
    }
}
```

```
status ListLength(SqList L) {
    // 如果线性表 L 存在, 返回线性表 L 的长度, 否则返回 INFEASIBLE。
    if (!L.elem)
        return INFEASIBLE;
    else
        return L.length;
}
```

```
status GetElem(SqList L, int i, ElemType &e) {
    // 如果线性表 L 存在, 获取线性表 L 的第 i 个元素, 保存在 e 中, 返回 OK;
    // 如果 i 不合法, 返回 ERROR; 如果线性表 L 不存在, 返回 INFEASIBLE。
```

```
if (!L.elem)
return INFEASIBLE;
else {
if (i < 1 || i > L.length)
return ERROR;
else {
e = L.elem[i - 1];
return OK;
}
}
}
```

```
int LocateElem(SqList L, ElemType e) {
// 如果线性表 L 存在，查找元素 e 在线性表 L 中的位置序号并返回该序号；
// 如果 e 不存在，返回 0；当线性表 L 不存在时，返回 INFEASIBLE（即-1）。
if (!L.elem)
return INFEASIBLE;
else {
int flag = -1;
for (int i = 0; i < L.length; i++)
if (L.elem[i] == e) {
// 遍历线性表中元素，若有相同就跳出
flag = i + 1;
break;
}
if (flag != -1)
return flag;
else
return ERROR;
}
}
```

```
status PriorElem(SqList L, ElemType e, ElemType &pre) {  
    // 如果线性表 L 存在, 获取线性表 L 中元素 e 的前驱, 保存在 pre 中, 返回 OK;  
    // 如果没有前驱, 返回 ERROR; 如果线性表 L 不存在, 返回 INFEASIBLE。  
    if (!L.elem)  
        return INFEASIBLE;  
    else {  
        int loc;  
        for (loc = 0; loc < L.length; loc++)  
            if (L.elem[loc] == e)  
                break; // 搜索元素位置  
        if (loc == 0 || loc == L.length)  
            return ERROR; // 元素位置为首位, 或线性表中不存在该元素  
        else {  
            pre = L.elem[loc - 1];  
            return OK;  
        }  
    }  
}
```

```
status NextElem(SqList L, ElemType e, ElemType &next) {  
    // 如果线性表 L 存在, 获取线性表 L 元素 e 的后继, 保存在 next 中, 返回 OK;  
    // 如果没有后继, 返回 ERROR; 如果线性表 L 不存在, 返回 INFEASIBLE。  
    if (!L.elem)  
        return INFEASIBLE;  
    else {  
        int loc;  
        for (loc = 0; loc < L.length; loc++)  
            if (L.elem[loc] == e)  
                break; // 搜索元素位置  
        if (loc == L.length - 1 || loc == L.length)  
            return ERROR;  
        else {  
            next = L.elem[loc + 1];  
            return OK;  
        }  
    }  
}
```

```
return ERROR;// 元素位置为末位，或线性表中不存在该元素
else {
    next = L.elem[loc + 1];
    return OK;
}
}
}
```

```
status ListInsert(SqList &L, int i, ElemType e) {
// 如果线性表 L 存在，将元素 e 插入到线性表 L 的第 i 个元素之前，返回 OK；
// 当插入位置不正确时，返回 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。
if (!L.elem)
    return INFEASIBLE;
else {
    if (i < 1 || i > L.length + 1)
        return ERROR;
    else {
        if (L.length == 0) {
            L.elem[0] = e;
            L.length++;
        } else {
            L.length++;
            if (L.length > L.listsize) {
                ElemType *newbase = (ElemType *)realloc(L.elem, (L.listsize + LISTINCREMENT)
                * sizeof(ElemType));
                L.elem = newbase;
                L.listsize += LISTINCREMENT;
            }
            // 如果 L.length 已经达到最大，则需要扩容
        }
        for (int j = L.length - 2; j >= i - 1; j--)
            L.elem[j + 1] = L.elem[j];
    }
}
```

```
// 将后面的元素全部后移一位
```

```
L.elem[i - 1] = e;
```

```
}
```

```
return OK;
```

```
}
```

```
}
```

```
}
```

```
status ListDelete(SqList &L, int i, ElemType &e) {
```

```
// 如果线性表 L 存在，删除线性表 L 的第 i 个元素，并保存在 e 中，返回 OK;
```

```
// 当删除位置不正确时，返回 ERROR; 如果线性表 L 不存在，返回 INFEASIBLE。
```

```
if (!L.elem)
```

```
return INFEASIBLE;
```

```
else {
```

```
if (i < 1 || i > L.length)
```

```
return ERROR;
```

```
else {
```

```
e = L.elem[i - 1];
```

```
for (int j = i - 1; j < L.length - 1; j++)
```

```
L.elem[j] = L.elem[j + 1];
```

```
// 将后面的元素全部前移一位
```

```
L.length--;
```

```
return OK;
```

```
}
```

```
}
```

```
}
```

```
status ListTraverse(SqList L) {
```

```
// 如果线性表 L 存在，依次显示线性表中的元素，每个元素间空一格，返回 OK;
```

```
// 如果线性表 L 不存在，返回 INFEASIBLE。
```

```
if (!L.elem)
```

```
return INFEASIBLE;
else {
    if (L.length > 0) {
        printf("\n-----all elements ----- \n");
        for (int i = 0; i < L.length - 1; i++)
            printf("%d ", L.elem[i]);
        printf("%d", L.elem[L.length - 1]);
        printf("\n----- end ----- \n");
    } else
        printf("线性表为空! \n");
    return OK;
}
}
```

```
status SaveList(SqList L, char FileName[]) {
    // 如果线性表 L 存在，将 L 的元素写到 FileName 文件中，返回 OK，否则返回
    INFEASIBLE。
    if (!L.elem)
        return INFEASIBLE;
    else {
        FILE *fp;
        fp = fopen(FileName, "w");
        for (int i = 0; i < L.length; i++)
            fprintf(fp, "%d ", L.elem[i]);
        // 不断读入数字，直到读入了 L.length 个
        fclose(fp);
        return OK;
    }
}
```

```
status LoadList(SqList &L, char FileName[]) {
```

// 如果线性表 L 不存在, 将 FileName 文件中的元素写入 L, 返回 OK, 否则返回 INFEASIBLE。

```
if (L.elem)
```

```
return INFEASIBLE;
```

```
else {
```

```
FILE *fp;
```

```
fp = fopen(FileName, "r");
```

```
L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
```

```
// 创建新的数据空间
```

```
int len = 0;
```

```
while (!feof(fp))
```

```
fscanf(fp, "%d", &L.elem[len++]);
```

```
// 从文件中写入数据到线性表中, 直到文件结束
```

```
L.length = len - 1;
```

```
fclose(fp);
```

```
return OK;
```

```
}
```

```
}
```

```
status AddList(LISTS &Lists, char ListName[]) {
```

```
// 在 Lists 中增加一个名称为 ListName 的空线性表
```

```
Lists.elem[Lists.length].L.elem = NULL;
```

```
Lists.elem[Lists.length].L.length = 0;
```

```
InitList(Lists.elem[Lists.length++].L);
```

```
// 线性表集合长度 +1
```

```
int t = 0;
```

```
while (ListName[t] != 0) {
```

```
Lists.elem[Lists.length - 1].name[t] = ListName[t];
```

```
t++;
```

```
}
```

```
return OK;
```

```
}
```

```
status RemoveList(LISTS &Lists, char ListName[]) {  
    // 在 Lists 中删除一个名称为 ListName 的线性表, 成功返回 OK, 否则返回 ERROR  
    if (Lists.length < 1)  
        return ERROR;  
    else {  
        int i, j, k, flag = 0;  
        for (i = 1; i <= Lists.length; i++) {  
            // 遍历线性表名称  
            for (j = 0, k = 0; Lists.elem[i - 1].name[j] != 0; j++) {  
                if (Lists.elem[i - 1].name[j] != ListName[j])  
                    break;  
                else  
                    k++;  
            }  
            if (j == k && j != 0) {  
                flag = 1;  
                break;  
            }  
        }  
        // 如果搜索成功则 flag 置为 1, 进行后续删除操作  
        if (flag == 1) {  
            DestroyList(Lists.elem[i - 1].L);  
            for (j = i; j < Lists.length; j++) {  
                Lists.elem[j - 1].L = Lists.elem[j].L;  
                k = 0;  
                while (Lists.elem[j].name[k] != 0) {  
                    Lists.elem[j - 1].name[k] = Lists.elem[j].name[k];  
                    k++;  
                }  
            }  
        }  
    }  
}
```



```
}  
// 后续线性表位置全部前移一位  
Lists.length--;  
return OK;  
} else  
return ERROR;  
}  
}  
  
int LocateList(LISTS Lists, char ListName[]) {  
// 在 Lists 中查找一个名为 ListName 的线性表，成功返回其序号，否则返回 0  
if (Lists.length < 1)  
return 0;  
else {  
int i, j, k, flag = 0;  
for (i = 1; i <= Lists.length; i++) {  
for (j = 0, k = 0; Lists.elem[i - 1].name[j] != 0; j++) {  
if (Lists.elem[i - 1].name[j] != ListName[j])  
break;  
else  
k++;  
}  
if (j == k && j != 0) {  
flag = 1;  
break;  
}  
}  
// 查找原理同上  
if (flag == 1) {  
return i;  
} else
```

```
return 0;
```

```
}
```

```
}
```

```
status MaxSubArray(SqList L) {
```

```
// 在线性表 L 中求其连续子数组最大和，并返回这个值
```

```
if (!L.elem)
```

```
return INFEASIBLE;
```

```
else {
```

```
if (!L.length)
```

```
return ERROR;
```

```
else {
```

```
int max = -9999999, tmax; //tmax 用来暂存
```

```
for (int i = 1; i <= L.length; i++)
```

```
// 按子数组中所含元素个数进行遍历
```

```
for (int j = 1; j <= L.length - i + 1; j++) {
```

```
tmax = 0;
```

```
for (int k = j; k <= j + i - 1; k++)
```

```
tmax += L.elem[k - 1];
```

```
if (tmax > max)
```

```
max = tmax; // 不断更新最大值
```

```
}
```

```
return max;
```

```
}
```

```
}
```

```
}
```

```
status SubArrayNum(SqList L, int k) {
```

```
// 在线性表 L 中搜索和为一定值的连续子数组，成功返回其个数，否则返回 ER-  
ROR
```

```
if (!L.elem)
```

```
return INFEASIBLE;
else {
    if (!L.length)
        return ERROR;
    else {
        int tmax, count = 0; // count 用来计数
        for (int i = 1; i <= L.length; i++)
            // 按子数组中所含元素个数进行遍历
            for (int j = 1; j <= L.length - i + 1; j++) {
                tmax = 0;
                for (int k = j; k <= j + i - 1; k++)
                    tmax += L.elem[k - 1];
                if (tmax == k)
                    count++;
            }
        return count;
    }
}

status SortList(SqList L) {
    // 对线性表 L 进行升序排序，成功返回 OK，否则返回 ERROR
    if (!L.elem)
        return INFEASIBLE;
    else {
        if (!L.length)
            return ERROR;
        else {
            int t;
            for (int i = 0; i < L.length - 1; i++)
                for (int j = 0; j < L.length - i - 1; j++)
```

```
if (L.elem[j] > L.elem[j + 1]) {
```

```
t = L.elem[j];
```

```
L.elem[j] = L.elem[j + 1];
```

```
L.elem[j + 1] = t;
```

```
}
```

```
// 使用冒泡排序
```

```
return OK;
```

```
}
```

```
}
```

```
}
```

```
status AdvancedSave(SqList L, char FileName[]) {
```

```
// 对线性表 L 进行文件的存取功能
```

```
if (!L.elem)
```

```
return INFEASIBLE;
```

```
else {
```

```
FILE *fp;
```

```
fp = fopen(FileName, "w");
```

```
fprintf(fp, "线性表表长为: %d\n", L.length);
```

```
fprintf(fp, "线性表存储方式为: 顺序存储 (数组)。\\n 其元素为: \\n");
```

```
fprintf(fp, "(首元素) ");
```

```
for (int i = 0; i < L.length; i++)
```

```
fprintf(fp, "%d ", L.elem[i]);
```

```
fprintf(fp, "(尾元素)");
```

```
fclose(fp);
```

```
return OK;
```

```
}
```

```
}
```

```
int main(void) {
```

```
SqList L;
```

```
L.elem = NULL;
int op = 1;
int i, e, pre, next, cur, testans, knum, choice;
char FileName[30], ListName[50];
LISTS LA;
LA.length = 0; // 对线性表集合进行初始化
while (op) {
    system("cls");
    printf("\n\n");
    printf("-----\n");
    printf(" 基于顺序存储结构的线性表实现 \n 实验演示系统（计算机 2110 班李嘉鹏） \n");
    printf("-----\n");
    printf(" 1. InitList 7. LocateElem\n");
    printf(" 2. DestroyList 8. PriorElem\n");
    printf(" 3. ClearList 9. NextElem \n");
    printf(" 4. ListEmpty 10. ListInsert\n");
    printf(" 5. ListLength 11. ListDelete\n");
    printf(" 6. GetElem 12. ListTraverse\n");
    printf(" 13. Save/Load 14. MaxSubArray\n");
    printf(" 15. SubArrayNum 16. SortList\n");
    printf(" 17. AdvancedFile 18. MultiList\n");
    printf(" 0. Exit\n");
    printf("-----\n");
    printf(" 请选择你的操作 [0 18]:\n");
    scanf("%d", &op);
    switch (op) {
        // 以下分别对应 18 个功能，根据输入情况执行对应的操作
        case 1:
            if (InitList(L) == OK)
                printf(" 线性表创建成功！ \n");
```

```
else
printf(" 线性表创建失败！ \n");
getchar();
getchar();
break;
case 2:
if (DestroyList(L) == OK)
printf(" 数据元素空间释放完成，线性表销毁成功！ \n");
else
printf(" 线性表销毁失败！ \n");
getchar();
getchar();
break;
case 3:
if (ClearList(L) == OK)
printf(" 线性表中所有数据元素已删除，线性表清空成功！ \n");
else
printf(" 线性表清空失败！ \n");
getchar();
getchar();
break;
case 4:
if (ListEmpty(L) == TRUE)
printf(" 线性表为空！ \n");
else if (ListEmpty(L) == FALSE)
printf(" 线性表不为空！ \n");
else if (ListEmpty(L) == INFEASIBLE)
printf(" 线性表不存在！ \n");
getchar();
getchar();
break;
```

```
case 5:
if (ListLength(L) == INFEASIBLE)
printf(" 线性表不存在! \n");
else
printf(" 线性表的长度为%d! \n", ListLength(L));
getchar();
getchar();
break;
case 6:
printf(" 请输入要查询的元素位置序号: ");
scanf("%d", &i);
testans = GetElem(L, i, e);
if (testans == OK)
printf(" 线性表的第%d 个元素为%d! \n", i, e);
else if (testans == ERROR)
printf(" 所给 i 值不合法! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 7:
printf(" 请输入要查询的元素的值: ");
scanf("%d", &e);
testans = LocateElem(L, e);
if (testans == ERROR)
printf(" 所给元素不存在! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
else
printf(" 元素%d 在线性表中的位置序号为%d! \n", e, testans);
```

```
getchar();
getchar();
break;
case 8:
printf(" 请输入要查询前驱的元素的值: ");
scanf("%d", &e);
testans = PriorElem(L, e, pre);
if (testans == OK)
printf(" 元素%d 的前驱为%d! \n", e, pre);
else if (testans == ERROR)
printf(" 该元素没有前驱! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 9:
printf(" 请输入要查询后继的元素的值: ");
scanf("%d", &e);
testans = NextElem(L, e, next);
if (testans == OK)
printf(" 元素%d 的后继为%d! \n", e, next);
else if (testans == ERROR)
printf(" 该元素没有后继! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 10:
printf(" 请输入插入位置和插入元素的值, 用空格隔开: ");
```



```
scanf("%d %d", &i, &e);
testans = ListInsert(L, i, e);
if (testans == OK)
printf(" 插入成功! \n");
else if (testans == ERROR)
printf(" 插入位置不正确! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 11:
printf(" 请输入要删除的元素的位置: ");
scanf("%d", &i);
testans = ListDelete(L, i, e);
if (testans == OK)
printf(" 删除成功, 删除的元素是%d! \n", e);
else if (testans == ERROR)
printf(" 删除位置不正确! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 12:
if (ListTraverse(L) == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 13:
```

```
printf(" 请输入目标文件名: ");
scanf("%s", FileName);
if (L.elem) {
testans = SaveList(L, FileName);
if (testans == OK)
printf(" 线性表中的元素已成功写入文件%s! \n", FileName);
else
printf(" 写文件失败! \n");
} else {
testans = LoadList(L, FileName);
if (testans == OK)
printf(" 文件%s 中的数据已成功读入到线性表中! \n", FileName);
else
printf(" 文件写入失败! \n");
}
getchar();
getchar();
break;
case 14:
if (MaxSubArray(L) == ERROR)
printf(" 线性表为空! \n");
else if (MaxSubArray(L) == INFEASIBLE)
printf(" 线性表不存在! \n");
else
printf(" 线性表中连续子数组的最大和是%d\n", MaxSubArray(L));
getchar();
getchar();
break;
case 15:
printf(" 请输入想要查找连续子数组的和: ");
scanf("%d", &knum);
```

```
if (SubArrayNum(L, knum) == ERROR)
printf(" 线性表中不存在这样的连续子数组! \n");
else if (SubArrayNum(L, knum) == INFEASIBLE)
printf(" 线性表不存在! \n");
else
printf(" 线性表中和为%d 的连续子数组的个数是%d\n", knum, SubArrayNum(L,
knum));
getchar();
getchar();
break;
case 16:
if (SortList(L) == ERROR)
printf(" 线性表为空! \n");
else if (SortList(L) == INFEASIBLE)
printf(" 线性表不存在! \n");
else
printf(" 线性表由小到大排序完成! \n");
getchar();
getchar();
break;
case 17:
printf(" 请输入目标文件名: ");
scanf("%s", FileName);
if (L.elem) {
testans = AdvancedSave(L, FileName);
if (testans == OK)
printf(" 线性表数据逻辑结构 (D,{R}) 完整信息已成功写入文件%s! \n", File-
Name);
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
else
```

```
printf(" 写文件失败! \n");
}
getchar();
getchar();
break;
case 18:
printf(" 请选择功能: 1. 线性表添加; 2. 线性表移除; 3. 线性表查找\n");
scanf("%d", &choice);
if (choice == 1) {
printf(" 请输入线性表名称: ");
scanf("%s", ListName);
if (AddList(LA, ListName) == OK)
printf(" 名为%s 的线性表已加入线性表集合中", ListName);
} else if (choice == 2) {
printf(" 请输入线性表名称: ");
scanf("%s", ListName);
if (RemoveList(LA, ListName) == OK)
printf(" 名为%s 的线性表已从线性表集合中删除", ListName);
} else if (choice == 3) {
printf(" 请输入线性表名称: ");
scanf("%s", ListName);
if (LocateList(LA, ListName) == 0)
printf(" 线性表集合中不存在这个线性表! ");
else
printf(" 该线性表的位置是%d", LocateList(LA, ListName));
}
getchar();
getchar();
break;
case 0:
break;
```

```
}  
}  
printf(" 欢迎下次再使用本系统! \n");  
return 0;  
}
```

## 附录 B 基于链式存储结构线性表实现的源程序

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>

//各类常量定义如下
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef int status; //数据元素类型定义
typedef int ElemType;

typedef struct LNode {
    ElemType data;
    struct LNode *next;
} LNode, *LinkList; // 链表节点结构及其指针定义

typedef struct {
    struct {
        char name[30];
        LinkList L;
    } elem[100];
    int length;
} LISTS; //链表集合结构定义
```

```
status InitList(LinkList &L)
```

```
// 线性表 L 不存在，构造一个空的线性表，返回 OK，否则返回 INFEASIBLE。
```

```
{  
if (L)  
return INFEASIBLE;  
else {  
L = (LinkList)malloc(sizeof(LNode));  
L->next = NULL;  
return OK;  
}  
}
```

```
status DestroyList(LinkList &L)
```

```
// 如果线性表存在，销毁线性表 L，释放数据元素空间，返回 OK，否则返回 INFEASIBLE。
```

```
{  
if (!L)  
return INFEASIBLE;  
else {  
while (L) {  
LinkList q;  
q = L->next;  
free(L);  
L = q;  
}  
L = NULL;  
return OK;  
}  
}
```

```
status ClearList(LinkList &L)
```

```
// 如果线性表 L 存在，删除线性表 L 中的所有元素，返回 OK，否则返回 INFEASIBLE。
```

```
{  
if (!L)  
return INFEASIBLE;  
else {  
LinkList q, p;  
p = L->next;  
while (p) {  
q = p->next;  
free(p);  
p = q;  
}  
L->next = NULL;  
return OK;  
}  
}
```

```
status ListEmpty(LinkList L)
```

```
// 如果线性表 L 存在，判断线性表 L 是否为空，空就返回 TRUE，否则返回 FALSE;
```

```
// 如果线性表 L 不存在，返回 INFEASIBLE。
```

```
{  
if (!L)  
return INFEASIBLE;  
else {  
if (L->next == NULL)  
return TRUE;  
else  
return FALSE;  
}  
}
```



```
}
```

```
int ListLength(LinkList L) {  
    // 如果线性表 L 存在，返回线性表 L 的长度，否则返回 INFEASIBLE。  
    if (!L)  
        return INFEASIBLE;  
    else {  
        int count = 0;  
        LinkList p = L;  
        while (p->next != NULL) {  
            p = p->next;  
            count++;  
        }  
        return count;  
    }  
}
```

```
status GetElem(LinkList L, int i, ElemType &e) {  
    // 如果线性表 L 存在，获取线性表 L 的第 i 个元素，保存在 e 中，返回 OK；  
    // 如果 i 不合法，返回 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。  
    if (!L)  
        return INFEASIBLE;  
    else {  
        if (i < 1)  
            return ERROR;  
        else {  
            int count = 0, flag = 0;  
            LinkList p = L;  
            while (p->next != NULL) {  
                p = p->next;  
                count++;  
            }  
        }  
    }  
}
```

```
if (count == i) {  
    flag = 1;  
    e = p->data;  
    break;  
}  
}  
if (flag == 0)  
    return ERROR;  
else  
    return OK;  
}  
}  
}
```

```
status LocateElem(LinkList L, ElemType e) {  
    // 如果线性表 L 存在，查找元素 e 在线性表 L 中的位置序号；  
    // 如果 e 不存在，返回 ERROR；当线性表 L 不存在时，返回 INFEASIBLE。  
    if (!L)  
        return INFEASIBLE;  
    else {  
        int count = 0, flag = 0;  
        LinkList p = L;  
        while (p->next != NULL) {  
            p = p->next;  
            count++;  
            if (p->data == e) {  
                flag = 1;  
                break;  
            }  
        }  
        if (flag == 0)
```

```
return ERROR;
else
return count;
}
}
```

```
status PriorElem(LinkList L, ElemType e, ElemType &pre) {
// 如果线性表 L 存在，获取线性表 L 中元素 e 的前驱，保存在 pre 中，返回 OK；
// 如果没有前驱，返回 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。
if (!L)
return INFEASIBLE;
else {
LinkList p = L;
int flag = 0;
while (p->next != NULL) {
p = p->next;
if (p->next != NULL && p->next->data == e) {
flag = 1;
pre = p->data;
break;
}
}
if (!flag)
return ERROR;
else
return OK;
}
}
```

```
status NextElem(LinkList L, ElemType e, ElemType &next) {
// 如果线性表 L 存在，获取线性表 L 元素 e 的后继，保存在 next 中，返回 OK；
```

// 如果没有后继, 返回 ERROR; 如果线性表 L 不存在, 返回 INFEASIBLE。

```
if (!L)
return INFEASIBLE;
else {
LinkList p = L;
int flag = 0;
while (p->next != NULL) {
p = p->next;
if (p->data == e && p->next != NULL) {
flag = 1;
next = p->next->data;
break;
}
}
if (!flag)
return ERROR;
else
return OK;
}
}
```

```
status ListInsert(LinkList &L, int i, ElemType e) {
```

// 如果线性表 L 存在, 将元素 e 插入到线性表 L ‘的第 i 个元素之前, 返回 OK;

// 当插入位置不正确时, 返回 ERROR; 如果线性表 L 不存在, 返回 INFEASIBLE。

```
if (!L)
return INFEASIBLE;
else {
LinkList p = L;
int pos = 0, flag;
while (p->next != NULL) {
pos++;
```

```
p = p->next;
}
if ((L->next != NULL && i < 1) || i > pos + 1)
flag = 0;
else {
pos = 0;
p = L;
while (pos < i - 1) {
pos++;
p = p->next;
}
LinkedList newnode = (LinkedList)malloc(sizeof(LNode));
newnode->data = e;
newnode->next = p->next;
p->next = newnode;
flag = 1;
}

if (flag)
return OK;
else
return ERROR;
}
}

status ListDelete(LinkedList &L, int i, ElemType &e) {
// 如果线性表 L 存在，删除线性表 L 的第 i 个元素，并保存在 e 中，返回 OK；
// 当删除位置不正确时，返回 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。
if (!L)
return INFEASIBLE;
else {
```

```
LinkedList p = L, q;
int pos = 0, flag = 1;
while (p->next != NULL) {
    pos++;
    p = p->next;
}
if (i < 1 || i > pos)
    flag = 0;
else {
    pos = 0;
    p = L;
    q = L->next;
    while (pos < i - 1) {
        pos++;
        p = p->next;
        q = q->next;
    }
    e = q->data;
    p->next = q->next;
    free(q);
}

if (flag)
    return OK;
else
    return ERROR;
}
}
```

```
status ListTraverse(LinkedList L) {
// 如果线性表 L 存在，依次显示线性表中的元素，每个元素间空一格，返回 OK;
```

// 如果线性表 L 不存在，返回 INFEASIBLE。

```
if (!L)
return INFEASIBLE;
else {
LinkedList p = L->next;
if (p != NULL) {
printf("\n————all elements —————\n");
while (p != NULL) {
printf("%d ", p->data);
p = p->next;
}
printf("\n———— end —————\n");
} else
printf(" 线性表为空! \n");
return OK;
}
}
```

status SaveList(LinkedList L, char FileName[]) {

// 如果线性表存在，将 L 的元素写到 FileName 文件中，返回 OK，否则返回 INFEASIBLE

```
if (!L)
return INFEASIBLE;
else {
FILE *fp;
fp = fopen(FileName, "w");
LinkedList p = L;
while (p->next != NULL) {
fprintf(fp, "%d ", p->next->data);
p = p->next;
}
}
```

```
fclose(fp);
return OK;
}
}
```

```
status LoadList(LinkList &L, char FileName[]) {
// 如果线性表不存在，将 FileName 的数据读入到 L 中，返回 OK，否则返回 IN-
FEASIBLE
if (L)
return INFEASIBLE;
else {
L = (LinkList)malloc(sizeof(LNode));
LinkList p = L;
FILE *fp;
fp = fopen(FileName, "r");
while (!feof(fp)) {
LinkList newnode = (LinkList)malloc(sizeof(LNode));
newnode->next = NULL;
p->next = newnode;
fscanf(fp, "%d", &newnode->data);
if (newnode->data == 0)
p->next = NULL;
p = p->next;
}
fclose(fp);
return OK;
}
}
```

```
status ReverseList(LinkList L)
// 反转链表 L，成功返回 OK；若链表不存在，返回 INFEASIBLE
```



```
{
if (!L)
return INFEASIBLE;
else {
if (L->next == NULL)
printf("链表不存在! \n");
else {
int temp[10000];
int count = 0;
LinkList p = L->next;
while (p != NULL) {
temp[++count] = p->data;
p = p->next;
}
p = L->next;
while (p != NULL) {
p->data = temp[count--];
p = p->next;
}
}
return OK;
}
}
```

status RemoveNthFromEnd(LinkList L, int n)

// 删除链表的倒数第 n 个结点，成功返回 OK；若链表不存在，返回 INFEASIBLE；

// 若 n 的值非法，返回 ERROR；若链表为空，返回-3

```
{
if (!L)
return INFEASIBLE;
else {
```

```
if (L->next == NULL)
return -3;
else {
int count = 0, i, e;
LinkedList p = L->next;
while (p != NULL) {
count++;
p = p->next;
}
if (n > count)
return ERROR;
else
return ListDelete(L, count - n + 1, e);
}
}
}
```

```
status sortList(LinkedList L)
// 将链表 L 由小到大排序，成功返回 OK；若链表不存在，返回 INFEASIBLE；
// 若链表为空，返回 ERROR
{
if (!L)
return INFEASIBLE;
else {
if (L->next == NULL)
return ERROR;
else {
int temp[1000];
int count = 0, t;
LinkedList p = L->next;
while (p != NULL) {
```

```
temp[++count] = p->data;
p = p->next;
}
p = L->next;
for (int i = 1; i < count; i++)
for (int j = 1; j <= count - i; j++)
if (temp[j + 1] < temp[j]) {
t = temp[j];
temp[j] = temp[j + 1];
temp[j + 1] = t;
}
count = 0;
while (p != NULL) {
p->data = temp[++count];
p = p->next;
}
return OK;
}
}
}
```

```
int main() {
LinkedList L;
L = NULL;
LISTS LLL;
LLL.length = 0;
int op = 1;
int i, j, e, testans, pre, next;
char FileName[30], ListName[30];
while (op) {
system("cls");
```

```
printf("\n\n");
printf("-----\n");
printf(" 基于链式存储结构的线性表实现 \n 实验演示系统（计算机 2110 班李嘉  
鹏） \n");
printf("-----\n");
printf(" 1. InitList 2. DestroyList\n");
printf(" 3. ClearList 4. ListEmpty\n");
printf(" 5. ListLength 6. GetElem\n");
printf(" 7. LocateElem 8. PriorElem\n");
printf(" 9. NextElem 10. ListInsert\n");
printf(" 11. ListDelete 12. ListTraverse\n");
printf(" 13. Save/Load 14. ReverseList\n");
printf(" 15. RemoveNthFromEnd 16. SortList\n");
printf(" 17. MultiList 0. Exit\n");
printf("-----\n");
printf(" 请选择操作 [0 17]:\n");
scanf("%d", &op);
switch (op) {
// 以下分别对应 17 个功能，根据输入情况执行对应的操作
case 1:
if (InitList(L) == OK)
printf(" 线性表创建成功！ \n");
else
printf(" 线性表创建失败！ \n");
getchar();
getchar();
break;
case 2:
if (DestroyList(L) == OK)
printf(" 数据元素空间释放完成，线性表销毁成功！ \n");
else
```

```
printf("线性表不存在！\n");
getchar();
getchar();
break;
case 3:
if (ClearList(L) == OK)
printf("线性表中所有数据元素已删除，线性表清空成功！\n");
else
printf("线性表不存在！\n");
getchar();
getchar();
break;
case 4:
if (ListEmpty(L) == TRUE)
printf("线性表为空！\n");
else if (ListEmpty(L) == FALSE)
printf("线性表不为空！\n");
else if (ListEmpty(L) == INFEASIBLE)
printf("线性表不存在！\n");
getchar();
getchar();
break;
case 5:
if (ListLength(L) == INFEASIBLE)
printf("线性表不存在！\n");
else
printf("线性表的长度为%d！\n", ListLength(L));
getchar();
getchar();
break;
case 6:
```

```
printf(" 请输入要查询的元素位置序号: ");
scanf("%d", &i);
testans = GetElem(L, i, e);
if (testans == OK)
printf(" 线性表的第%d 个元素为%d! \n", i, e);
else if (testans == ERROR)
printf(" 所给 i 值不合法! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 7:
printf(" 请输入要查询的元素的值: ");
scanf("%d", &e);
testans = LocateElem(L, e);
if (testans == ERROR)
printf(" 所给元素不存在! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
else
printf(" 元素%d 在线性表中的位置序号为%d! \n", e, testans);
getchar();
getchar();
break;
case 8:
printf(" 请输入要查询前驱的元素的值: ");
scanf("%d", &e);
testans = PriorElem(L, e, pre);
if (testans == OK)
printf(" 元素%d 的前驱为%d! \n", e, pre);
```

```
else if (testans == ERROR)
printf(" 该元素没有前驱! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 9:
printf(" 请输入要查询后继的元素的值: ");
scanf("%d", &e);
testans = NextElem(L, e, next);
if (testans == OK)
printf(" 元素%d 的后继为%d! \n", e, next);
else if (testans == ERROR)
printf(" 该元素没有后继! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 10:
printf(" 请输入插入位置和插入元素的值, 用空格隔开: ");
scanf("%d %d", &i, &e);
testans = ListInsert(L, i, e);
if (testans == OK)
printf(" 插入成功! \n");
else if (testans == ERROR)
printf(" 插入位置不正确! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
```

```
getchar();
break;
case 11:
printf(" 请输入要删除的元素的位置: ");
scanf("%d", &i);
testans = ListDelete(L, i, e);
if (testans == OK)
printf(" 删除成功, 删除的元素是%d! \n", e);
else if (testans == ERROR)
printf(" 删除位置不正确! \n");
else if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 12:
if (ListTraverse(L) == INFEASIBLE)
printf(" 线性表不存在! \n");
getchar();
getchar();
break;
case 13:
printf(" 请输入目标文件名: ");
scanf("%s", FileName);
if (L) {
testans = SaveList(L, FileName);
if (testans == OK)
printf(" 线性表中的元素已成功写入文件%s! \n", FileName);
else
printf(" 写文件失败! \n");
} else {
```



```
testans = LoadList(L, FileName);
if (testans == OK)
printf(" 文件%s 中的数据已成功读入到线性表中! \n", FileName);
else
printf(" 文件写入失败! \n");
}
getchar();
getchar();
break;
case 14:
testans = ReverseList(L);
if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
else if (L->next != NULL)
printf(" 链表反转完成! \n");
getchar();
getchar();
break;
case 15:
printf(" 请输入想要移除的元素是倒数第几位: ");
int choice;
scanf("%d", &choice);
testans = RemoveNthFromEnd(L, choice);
if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
else if (testans == ERROR)
printf(" 所给位置不合法! \n");
else if (testans == -3)
printf(" 链表为空! \n");
else
printf(" 该元素已移除! \n");
```

```
getchar();
getchar();
break;
case 16:
testans = sortList(L);
if (testans == INFEASIBLE)
printf(" 线性表不存在! \n");
else if (testans == ERROR)
printf(" 链表为空! \n");
else
printf(" 链表已从小到大排序! \n");
getchar();
getchar();
break;
case 17:
printf(" 请选择功能: 1. 线性表添加; 2. 线性表移除; 3. 线性表查找\n");
scanf("%d", &choice);
if (choice == 1) {
printf(" 请输入线性表名称: ");
scanf("%s", LLL.elem[LLL.length].name);
printf(" 请输入线性表%s 的元素个数和元素值, 以空格隔开: ");
LLL.elem[LLL.length].L = NULL;
int count;
int tem[100];
scanf("%d", &count);
for (int i = 1; i <= count; i++)
scanf("%d", &tem[i]);
InitList(LLL.elem[LLL.length].L);
// LLL.elem[LLL.length].L->next = NULL;
for (int i = 1; i <= count; i++)
ListInsert(LLL.elem[LLL.length].L, 1, tem[i]);
```

```
ReverseList(LLL.elem[LLL.length].L);
LLL.length++;
printf(" 名为%s 的线性表插入成功! \n", LLL.elem[LLL.length - 1].name);
} else if (choice == 2) {
    if (LLL.length <= 0)
        printf(" 删除失败, 线性表集合为空! \n");
    else {
        printf(" 请输入线性表名称: ");
        scanf("%s", ListName);
        int flag = 0;
        for (i = 0; i < LLL.length; i++)
            if (strcmp(LLL.elem[i].name, ListName) == 0) {
                flag = 1;
                break;
            }
        if (flag) {
            for (j = i; j = LLL.length - 1; j++) {
                LLL.elem[j].L = LLL.elem[j + 1].L;
                strcpy(LLL.elem[j].name, LLL.elem[j + 1].name);
            }
            LLL.length--;
            printf(" 名为%s 的线性表已从线性表集合中删除", ListName);
        } else
            printf(" 线性表集合中没有名为%s 的线性表! \n", ListName);
    }
} else if (choice == 3) {
    if (LLL.length <= 0)
        printf(" 线性表集合为空! \n");
    else {
        printf(" 请输入线性表名称: ");
        scanf("%s", ListName);
```

```
int flag = 0;
for (i = 0; i < LLL.length; i++)
if (strcmp(LLL.elem[i].name, ListName) == 0) {
flag = 1;
break;
}
if (flag) {
InitList(L);
L = LLL.elem[i].L;
printf(" 查找成功! \n");
} else
printf(" 线性表集合中没有名为%s 的线性表! \n", ListName);
}
} else
printf(" 请重新选择! \n");
getchar();
getchar();
break;
case 0:
break;
}
}
printf(" 欢迎下次再使用本系统! \n");
return 0;
}
```

## 附录 C 基于二叉链表二叉树实现的源程序

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>

// 各类常量定义如下
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

typedef int status;
typedef int KeyType;
typedef struct {
    KeyType key;
    char others[20];
} TElemType; //二叉树结点类型定义

typedef struct BiTNode { //二叉链表结点定义
    TElemType data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

typedef struct {
    struct {
        char name[30];
        BiTree T;
```

```
} elem[100];
int length;
} TREES; //二叉树集合结构定义

int count = 0; //统计已经创建了几个结点
int dep[1000] = {0};

BiTree CreateBiTNode(TElemType definition[]) {
// 创建一个新的二叉链表结点
if (definition[count].key) {
BiTree newnode = (BiTree)malloc(sizeof(BiTNode));
newnode->data = definition[count++];
newnode->lchild = CreateBiTNode(definition);
newnode->rchild = CreateBiTNode(definition);
return newnode;
} else {
count++;
return NULL;
}
}

status CreateBiTree(BiTree &T, TElemType definition[])
/* 根据带空枝的二叉树先根遍历序列 definition 构造一棵二叉树，将根节点指针
赋值给 T 并返回 OK，
如果有相同的关键字，返回 ERROR。此题允许通过增加其它函数辅助实现本关
任务 */
{
if (T)
return INFEASIBLE;
int check[1000];
for (int i = 0; definition[i].key != -1; i++) {
```

```
if (definition[i].key != 0 && check[definition[i].key] == 1)
return ERROR;
check[definition[i].key] = 1;
}
count = 0;
T = CreateBiTNode(definition);
return OK;
}
```

```
status ClearBiTree(BiTree &T)
//将二叉树设置成空，并删除所有结点，释放结点空间
{
if (T) {
if (T->lchild) {
ClearBiTree(T->lchild);
}
if (T->rchild) {
ClearBiTree(T->rchild);
}
free(T);
T = NULL;
return OK;
} else
return INFEASIBLE;
}
```

```
int BiTreeDepth(BiTree T)
//求二叉树 T 的深度
{
if (T == NULL)
return 0;
```

```
else {
if (T->lchild != NULL && T->rchild != NULL)
return (BiTreeDepth(T->lchild) > BiTreeDepth(T->rchild) ? BiTreeDepth(T->lchild) :
BiTreeDepth(T->rchild)) + 1;
else if (T->lchild == NULL && T->rchild != NULL)
return BiTreeDepth(T->rchild) + 1;
else if (T->rchild == NULL && T->lchild != NULL)
return BiTreeDepth(T->lchild) + 1;
else
return 1;
}
}
```

BiTNode \*LocateNode(BiTree T, KeyType e)

//查找结点

```
{
if (!T)
return NULL;
else {
BiTree temp;
if (e == T->data.key)
return T;
else {
if (T->lchild) {
temp = LocateNode(T->lchild, e);
if (temp)
return temp;
}
if (T->rchild) {
temp = LocateNode(T->rchild, e);
if (temp)
```



```
return temp;
}
return NULL;
}
}
}
```

```
status check(BiTree T, KeyType newvalue, KeyType key)
```

```
// 检查新的赋值是否会产生冲突
```

```
{
int r = 1;
if (T) {
if (T->data.key == newvalue && T->data.key != key) {
r = 0;
return r;
}
r = check(T->lchild, newvalue, key);
if (!r)
return r;
r = check(T->rchild, newvalue, key);
if (!r)
return r;
}
return r;
}
```

```
status Assign(BiTree &T, KeyType e, TElemType value)
```

```
//实现结点赋值。此题允许通过增加其它函数辅助实现本关任务
```

```
{
BiTree ans;
ans = LocateNode(T, e);
```

```
if (ans) {
    if (check(T, value.key, e)) {
        ans->data = value;
        return OK;
    } else
        return ERROR;// 复制后关键字不唯一
    } else
        return ERROR;
}
```

```
BiTNode *GetSibling(BiTree T, KeyType e)
//实现获得兄弟结点
{
    if (T->lchild && T->lchild->data.key == e) {
        if (T->rchild)
            return T->rchild;
    } else if (T->rchild && T->rchild->data.key == e) {
        if (T->lchild)
            return T->lchild;
    } else {
        BiTree temp;
        if (T->lchild) {
            temp = GetSibling(T->lchild, e);
            if (temp)
                return temp;
        }
        if (T->rchild) {
            temp = GetSibling(T->rchild, e);
            if (temp)
                return temp;
        }
    }
}
```

```
}
```

```
return NULL;
```

```
}
```

```
status check1(BiTree T, KeyType newvalue)
```

```
// 检查关键字是否冲突，若返回 0 则说明冲突
```

```
{
```

```
int r = 1;
```

```
if (T) {
```

```
if (T->data.key == newvalue) {
```

```
    r = 0;
```

```
    return r;
```

```
}
```

```
r = check1(T->lchild, newvalue);
```

```
if (!r)
```

```
    return r;
```

```
r = check1(T->rchild, newvalue);
```

```
if (!r)
```

```
    return r;
```

```
}
```

```
return r;
```

```
}
```

```
status InsertNode(BiTree &T, KeyType e, int LR, TElemType c)
```

```
//插入结点。此题允许通过增加其它函数辅助实现本关任务
```

```
{
```

```
BiTree target, newnode;
```

```
if (LR == -1) { // c 作为根结点插入，原根结点作为 c 的右子树
```

```
BiTree newnode = (BiTree)malloc(sizeof(BiTNode));
```

```
newnode->data = c;
```

```
newnode->rchild = T;
```

```
newnode->lchild = NULL;
T = newnode;
return OK;
} else {
    target = LocateNode(T, e);
    if (!target)
        return ERROR;
    else if (!check1(T, c.key))
        return ERROR;
    else {
        if (LR == 0) { // 插入结点 c 到 T 中，作为关键字为 e 的结点的左孩子
            if (!T)
                return INFEASIBLE;
            BiTree newnode = (BiTree)malloc(sizeof(BiTNode));
            newnode->data = c;
            newnode->rchild = target->lchild;
            target->lchild = newnode;
            newnode->lchild = NULL;
            return OK;
        } else { // 插入结点 c 到 T 中，作为关键字为 e 的结点的右孩子
            if (!T)
                return INFEASIBLE;
            BiTree newnode = (BiTree)malloc(sizeof(BiTNode));
            newnode->data = c;
            newnode->rchild = target->rchild;
            target->rchild = newnode;
            newnode->lchild = NULL;
            return OK;
        }
    }
}
```

```
}
```

```
BiTNode *LocateParent(BiTree T, KeyType e) {  
    if (!T)  
        return NULL;  
    else {  
        if ((T->lchild && T->lchild->data.key == e) || (T->rchild && T->rchild->data.key ==  
            e))  
            return T;  
        else {  
            BiTree temp = NULL;  
            temp = LocateParent(T->lchild, e);  
            if (temp)  
                return temp;  
            temp = NULL;  
            temp = LocateParent(T->rchild, e);  
            return temp;  
        }  
    }  
}
```

```
status DeleteNode(BiTree &T, KeyType e)  
//删除结点。此题允许通过增加其它函数辅助实现本关任务  
{  
    BiTree target = LocateNode(T, e);  
    BiTree temp, parent;  
    if (!target)  
        return ERROR;  
    else {  
        if (target->lchild == NULL && target->rchild == NULL) {  
            parent = LocateParent(T, e);
```

```
if (parent) {
if (parent->lchild == target)
parent->lchild = NULL;
else
parent->rchild = NULL;
} else // 删除的点是根节点
T = temp;
free(target);
} else if (target->lchild != NULL && target->rchild == NULL) {
temp = target->lchild;
parent = LocateParent(T, e);
if (parent) {
if (parent->lchild == target)
parent->lchild = temp;
else
parent->rchild = temp;
} else
T = temp;
free(target);
} else if (target->rchild != NULL && target->lchild == NULL) {
temp = target->rchild;
parent = LocateParent(T, e);
if (parent) {
if (parent->lchild == target)
parent->lchild = temp;
else
parent->rchild = temp;
} else
T = temp;
free(target);
} else {
```

```
parent = LocateParent(T, e);
temp = target->lchild;
if (parent) {
    if (parent->lchild == target)
        parent->lchild = temp;
    if (parent->rchild == target)
        parent->rchild = temp;
} else
    T = temp;
temp = target->lchild;
while (temp->rchild)
    temp = temp->rchild;
temp->rchild = target->rchild;
free(target);
}
return OK;
}
}

void visit(BiTree T) {
    printf(" %d,%s", T->data.key, T->data.others);
}

status PreOrderTraverse(BiTree T, void (*visit)(BiTree))
//先序遍历二叉树 T
{
    if (T) {
        visit(T);
        PreOrderTraverse(T->lchild, visit);
        PreOrderTraverse(T->rchild, visit);
    }
}
```

```
return OK;
```

```
}
```

```
int top = 0; // 栈顶指针
```

```
BiTree stack[1000];
```

```
status InOrderTraverse(BiTree T, void (*visit)(BiTree))
```

```
//中序遍历二叉树 T
```

```
{
```

```
BiTree p = T;
```

```
while (p || top) {
```

```
if (p) { // 根指针进栈，遍历左子树
```

```
stack[top++] = p;
```

```
p = p->lchild;
```

```
} else { // 根指针退栈，访问根节点并遍历右子树
```

```
p = stack[--top];
```

```
visit(p);
```

```
p = p->rchild;
```

```
}
```

```
}
```

```
return OK;
```

```
}
```

```
status PostOrderTraverse(BiTree T, void (*visit)(BiTree))
```

```
//后序遍历二叉树 T
```

```
{
```

```
if (T) {
```

```
PostOrderTraverse(T->lchild, visit);
```

```
PostOrderTraverse(T->rchild, visit);
```

```
visit(T);
```

```
}
```



```
return OK;
```

```
}
```

```
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree))
```

```
//按层遍历二叉树 T
```

```
{
```

```
BiTree que[1000];
```

```
int front = 0;
```

```
int rear = 1;
```

```
if (T) {
```

```
que[0] = T;
```

```
while (rear - front > 0) {
```

```
visit(que[front]);
```

```
front++;
```

```
if (que[front - 1]->lchild)
```

```
que[rear++] = que[front - 1]->lchild;
```

```
if (que[front - 1]->rchild)
```

```
que[rear++] = que[front - 1]->rchild;
```

```
}
```

```
}
```

```
return OK;
```

```
}
```

```
status Save(BiTree T, FILE *fp)
```

```
// 先序遍历存储
```

```
{
```

```
if (T) {
```

```
fprintf(fp, "%d %s\n", T->data.key, T->data.others);
```

```
Save(T->lchild, fp);
```

```
Save(T->rchild, fp);
```

```
} else
```

```
fprintf(fp, "0 null\n");  
return OK;  
}
```

```
BiTree Load(FILE *fp)  
// 先序遍历读取  
{  
    BiTree T;  
    TElemType load;  
    fscanf(fp, "%d %s", &load.key, load.others);  
    if (load.key == 0)  
        T = NULL;  
    else {  
        T = (BiTree)malloc(sizeof(BiTNode));  
        T->data = load;  
        T->lchild = Load(fp);  
        T->rchild = Load(fp);  
    }  
    return T;  
}
```

```
status SaveBiTree(BiTree T, char FileName[])  
//将二叉树的结点数据写入到文件 FileName 中  
{  
    if (!T)  
        return INFEASIBLE;  
    else {  
        FILE *fp;  
        fp = fopen(FileName, "w");  
        Save(T, fp);  
        fclose(fp);  
    }
```

```
return OK;
}
}

status LoadBiTree(BiTree &T, char FileName[])

//读入文件 FileName 的结点数据，创建二叉树
{
if (T)
return INFEASIBLE;
else {
FILE *fp;
fp = fopen(FileName, "r");
T = Load(fp);
fclose(fp);
return OK;
}
}

status MaxPathSum(BiTree &T, int sum)
// 返回根节点到叶子结点的最大路径和
{
if (!T->lchild && !T->rchild)
return sum;
else {
if (T->lchild && T->rchild)
return MaxPathSum(T->lchild, sum + T->lchild->data.key) > MaxPathSum(T->rchild,
sum + T->rchild->data.key) ? MaxPathSum(T->lchild, sum + T->lchild->data.key) :
MaxPathSum(T->rchild,
sum + T->rchild->data.key);
else if (!T->lchild)
return MaxPathSum(T->lchild, sum + T->lchild->data.key);
```

```
else
return MaxPathSum(T->rchild, sum + T->rchild->data.key);
}
}
```

```
void GetDepth(BiTree &T, int predep) {
dep[T->data.key] = predep++;
if (T->lchild)
GetDepth(T->lchild, predep);
if (T->rchild)
GetDepth(T->rchild, predep);
}
```

```
BiTree LowestCommonAncestor(BiTree &T, KeyType e1, KeyType e2)
// 返回两个结点的最近公共祖先
{
BiTree m, n;
m = LocateNode(T, e1);
n = LocateNode(T, e2);
dep[1000] = {0};
GetDepth(T, 1);
if (dep[m->data.key] <= dep[n->data.key])
return LocateParent(T, e1);
else
return LocateParent(T, e2);
}
```

```
void InvertTree(BiTree &T)
// 翻转二叉树
{
if (T->lchild && T->rchild) {
```

```
BiTree temp;
temp = T->lchild;
T->lchild = T->rchild;
T->rchild = temp;
}
if (T->lchild)
InvertTree(T->lchild);
if (T->rchild)
InvertTree(T->rchild);
}

int main() {
int op = 1;
int i, result, e, LR, e1, e2, choice, j;
char FileName[30], TreeName[30];
TElemType value, c;
BiTree resultnode;
BiTree T = NULL;
TElemType definition[100];
TREES TTT;
TTT.length = 0;
while (op) {
system("cls");
printf("\n\n");
printf("-----\n");
printf(" 二叉树操作演示系统 \n 计算机 2110 班李嘉鹏\n");
printf("-----\n");
printf(" 1. CreateBitree 2. ClearBiTree\n");
printf(" 3. BiTreeDepth 4. LocateNode\n");
printf(" 5. Assign 6. GetSibling\n");
printf(" 7. InsertNode 8. DeleteNode\n");
```

```
printf(" 9. PreOrderTraverse 10. InOrderTraverse\n");
printf(" 11. PostOrderTraverse 12. LevelOrderTraverse\n");
printf(" 13. Save/Load 14. MaxPathSum\n");
printf(" 15. LowestCommonAncestor 16. InvertTree\n");
printf(" 17. MultiTree 0. Exit\n");
printf("-----\n");
printf(" 请选择操作 [0 17]:\n");
scanf("%d", &op);
switch (op) {
// 以下分别对应 17 个功能，根据输入情况执行对应的操作
case 1:
printf(" 请输入带空枝的二叉树先根遍历序列，以-1 null 结束: \n");
i = 0;
do {
scanf("%d%s", &definition[i].key, definition[i].others);
} while (definition[i++].key != -1);
if (CreateBiTree(T, definition) == OK)
printf(" 二叉树创建成功! \n");
else if (CreateBiTree(T, definition) == ERROR)
printf(" 存在相同关键字，二叉树创建失败! \n");
else
printf(" 已存在二叉树，创建失败! \n");
getchar();
getchar();
break;
case 2:
result = ClearBiTree(T);
if (result == INFEASIBLE)
printf(" 二叉树不存在! \n");
if (result == OK)
printf(" 二叉树清空成功! \n");
```

```
getchar();
getchar();
break;
case 3:
result = BiTreeDepth(T);
if (!result)
printf(" 二叉树不存在 (深度为 0)! \n");
else
printf(" 二叉树的深度为%d! \n", result);
getchar();
getchar();
break;
case 4:
if (!T)
printf(" 二叉树不存在! \n");
else {
printf(" 请输入要查找结点的关键字: ");
scanf("%d", &e);
resultnode = LocateNode(T, e);
if (!resultnode)
printf(" 二叉树中不存在关键字为%d 的结点! \n", e);
else
printf(" 已找到关键字为%d 的结点, 结点值为%s! \n", e, resultnode->data.others);
}
getchar();
getchar();
break;
case 5:
if (!T)
printf(" 二叉树不存在! \n");
else {
```

```
printf(" 请输入要赋值的结点关键字，以及新结点信息（关键字、结点值），用空格隔开： ");
scanf("%d %d %s", &e, &value.key, value.others);
result = Assign(T, e, value);
if (result == OK)
printf(" 赋值成功！ \n");
else
printf(" 赋值失败！ \n");
}
getchar();
getchar();
break;
case 6:
if (!T)
printf(" 二叉树不存在！ \n");
else {
printf(" 请输入要查找兄弟结点的结点的关键字： ");
scanf("%d", &e);
resultnode = GetSibling(T, e);
if (!result)
printf(" 关键字为%d 的结点不存在兄弟结点！ \n", e);
else
printf(" 关键字为%d 的结点存在兄弟结点,结点值为%s!\n", e, resultnode->data.others);
}
getchar();
getchar();
break;
case 7:
if (!T)
printf(" 二叉树不存在！ \n");
else {
```



```
printf(" 请输入目标结点的关键字、LR 和新结点信息（关键字、结点值），用空格隔开\n");
printf("（LR 为-1/0/1 分别代表作为根结点或作为目标结点的左或右孩子插入）：\n");
scanf("%d %d %d %s", &e, &LR, &c.key, c.others);
result = InsertNode(T, e, LR, c);
if (result == OK)
printf(" 插入成功！ \n");
else
printf(" 插入失败！ \n", result);
}
getchar();
getchar();
break;
case 8:
if (!T)
printf(" 二叉树不存在！ \n");
else {
printf(" 请输入要删除结点的关键字： ");
scanf("%d", &e);
result = DeleteNode(T, e);
if (result == OK)
printf(" 删除成功！ \n");
if (result == ERROR)
printf(" 删除失败！ \n", result);
}
getchar();
getchar();
break;
case 9:
if (!T)
```

```
printf(" 二叉树不存在! \n");
else {
printf("\n—————先序遍历结果—————\n");
result = PreOrderTraverse(T, visit);
if (result == OK)
printf("\n\n—————-end—————\n");
else
printf(" 遍历失败! \n", result);
}
getchar();
getchar();
break;
case 10:
if (!T)
printf(" 二叉树不存在! \n");
else {
printf("\n—————中序遍历结果—————\n");
result = InOrderTraverse(T, visit);
if (result == OK)
printf("\n\n—————-end—————\n");
else
printf(" 遍历失败! \n", result);
}
getchar();
getchar();
break;
case 11:
if (!T)
printf(" 二叉树不存在! \n");
else {
printf("\n—————后序遍历结果—————\n");
```

```
result = PostOrderTraverse(T, visit);
if (result == OK)
printf("\n\n-----end-----\n");
else
printf(" 遍历失败! \n", result);
}
getchar();
getchar();
break;
case 12:
if (!T)
printf(" 二叉树不存在! \n");
else {
printf("\n-----层序遍历结果-----\n");
result = LevelOrderTraverse(T, visit);
if (result == OK)
printf("\n\n-----end-----\n");
else
printf(" 遍历失败! \n", result);
}
getchar();
getchar();
break;
case 13:
printf(" 请输入目标文件名: ");
scanf("%s", FileName);
if (T) {
result = SaveBiTree(T, FileName);
if (result == OK)
printf(" 二叉树数据已成功写入文件%s! \n", FileName);
else
```

```
printf(" 写文件失败! \n");
} else {
result = LoadBiTree(T, FileName);
if (result == OK)
printf(" 文件%s 中的数据已成功读入到二叉树中! \n", FileName);
else
printf(" 文件写入失败! \n");
}
getchar();
getchar();
break;
case 14:
if (!T)
printf(" 二叉树不存在! \n");
else {
printf(" 根节点到叶子结点的最大路径和为%d! ", MaxPathSum(T, 0));
}
getchar();
getchar();
break;
case 15:
if (!T)
printf(" 二叉树不存在! \n");
else {
printf(" 请输入两个结点的关键字，用空格隔开: ");
scanf("%d %d", &e1, &e2);
resultnode = LowestCommonAncestor(T, e1, e2);
if (resultnode == NULL)
printf(" 这两个结点没有公共祖先! \n");
else
printf(" 这两个结点最近公共祖先的结点值为%s", resultnode->data.others);
```

```
}
getchar();
getchar();
break;
case 16:
if (!T)
printf(" 二叉树不存在! \n");
else {
InvertTree(T);
printf(" 二叉树翻转完成! \n");
}
getchar();
getchar();
break;
case 17:
printf(" 请选择功能: 1. 二叉树添加; 2. 二叉树移除; 3. 二叉树查找\n");
scanf("%d", &choice);
if (choice == 1) {
printf(" 请输入二叉树名称: ");
scanf("%s", TTT.elem[TTT.length].name);
TTT.elem[TTT.length].T = NULL;
printf(" 请输入二叉树%s 带空枝的先根遍历序列, 以-1 null 结束: ");
i = 0;
TElemType newdefinition[100];
do {
scanf("%d%s", &newdefinition[i].key, newdefinition[i].others);
} while (newdefinition[i++].key != -1);
count = 0;
if (CreateBiTree(TTT.elem[TTT.length].T, newdefinition) == OK) {
TTT.length++;
printf(" 名为%s 的二叉树添加成功! \n", TTT.elem[TTT.length - 1].name);
```

```
}
} else if (choice == 2) {
if (TTT.length <= 0)
printf(" 删除失败， 二叉树集合为空！ \n");
else {
printf(" 请输入二叉树名称： ");
scanf("%s", TreeName);
int flag = 0;
for (j = 0; j < TTT.length; j++)
if (strcmp(TTT.elem[j].name, TreeName) == 0) {
flag = 1;
break;
}
if (flag) {
for (int k = j; k < TTT.length - 1; k++) {
TTT.elem[k].T = TTT.elem[k + 1].T;
strcpy(TTT.elem[k].name, TTT.elem[k + 1].name);
}
TTT.length--;
printf(" 名为%s 的二叉树已从二叉树集合中删除！ ", TreeName);
} else
printf(" 二叉树集合中没有名为%s 的二叉树！ \n", TreeName);
}
} else if (choice == 3) {
if (TTT.length <= 0)
printf(" 二叉树集合为空！ \n");
else {
printf(" 请输入二叉树名称： ");
scanf("%s", TreeName);
int flag = 0;
for (i = 0; i < TTT.length; i++)
```

```
if (strcmp(TTT.elem[i].name, TreeName) == 0) {
    flag = 1;
    break;
}
if (flag) {
    T = TTT.elem[i].T;
    printf(" 查找成功! \n");
} else
    printf(" 二叉树集合中没有名为%s 的二叉树! \n", TreeName);
}
} else
    printf(" 请重新选择! \n");
    getchar();
    getchar();
    break;
case 0:
    break;
}
}
printf(" 欢迎下次再使用本系统! \n");
return 0;
}
```

## 附录 D 基于邻接表图实现的源程序

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>

// 各类常量定义如下
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define MAX_VERTEX_NUM 20 //最大顶点数
#define INF 999999

typedef int status;
typedef int KeyType;
typedef enum {DG, DN, UDG, UDN} GraphKind;

typedef struct {
    KeyType key;
    char others[20];
} VertexType; //顶点类型定义

typedef struct ArcNode { //表结点类型定义
    int adjvex; //顶点位置编号
    struct ArcNode *nextarc; //下一个表结点指针
} ArcNode;
```



```
typedef struct VNode { //头结点及其数组类型定义
```

```
VertexType data; //顶点信息
```

```
ArcNode *firstarc; //指向第一条弧
```

```
} VNode, AdjList[MAX_VERTEX_NUM];
```

```
typedef struct { //邻接表的类型定义
```

```
AdjList vertices; //头结点数组
```

```
int vexnum, arcnum; //顶点数、弧数
```

```
GraphKind kind; //图的类型
```

```
} ALGraph;
```

```
typedef struct {
```

```
struct {
```

```
char name[30];
```

```
ALGraph G;
```

```
} elem[100];
```

```
int length;
```

```
} GRAPHS; //无向图集合结构定义
```

```
status CreateCraph(ALGraph &G, VertexType V[], KeyType VR[][2])
```

```
/* 根据 V 和 VR 构造图 T 并返回 OK，如果 V 和 VR 不正确，返回 ERROR
```

```
如果有相同的关键字，返回 ERROR。此题允许通过增加其它函数辅助实现本关任务 */
```

```
{
```

```
int check[100] = {0}, order[100];
```

```
G.vexnum = 0, G.arcnum = 0;
```

```
for (int i = 0; V[i].key != -1; i++) {
```

```
if (check[V[i].key] == 1)
```

```
return ERROR; // 判断顶点关键字是否重复
```

```
check[V[i].key] = 1;
```

```
order[V[i].key] = i;
```

```
}
for (int i = 0; V[i].key != -1; i++) {
    G.vertices[i].data.key = V[i].key;
    strcpy(G.vertices[i].data.others, V[i].others);
    G.vertices[i].firstarc = NULL;
    G.vexnum++;
}
if (G.vexnum > 20 || G.vexnum < 1)
    return ERROR;
int count = 0, flag;
for (int i = 0; VR[i][0] != -1; i++) {
    if (check[VR[i][0]] != 1 || check[VR[i][1]] != 1)
        return ERROR; // 判断边的信息是否合法
    count++; // count 代表总共有几条边的信息
}
for (int i = count; i >= 0; i--) {
    if (VR[i][0] != VR[i][1]) {
        flag = 0;
        for (int j = 0; j < i; j++)
            if ((VR[j][0] == VR[i][0] && VR[j][1] == VR[i][1]) || (VR[j][1] == VR[i][0] &&
                VR[j][0] == VR[i][1])) {
                flag = 1;
                break;
            }
        if (flag != 1) {
            ArcNode *newnode = (ArcNode *)malloc(sizeof(ArcNode));
            newnode->adjvex = order[VR[i][1]]; // 创建第一个结点
            newnode->nextarc = NULL;
            if (check[VR[i][0]] == 1) {
                G.vertices[order[VR[i][0]]].firstarc = newnode;
                check[VR[i][0]] = 0;
            }
        }
    }
}
```

```
} else {
ArcNode *temp;
temp = G.vertices[order[VR[i][0]]].firstarc;
while (temp->nextarc)
temp = temp->nextarc;
temp->nextarc = newnode;
}
newnode = (ArcNode *)malloc(sizeof(ArcNode));
newnode->adjvex = order[VR[i][0]]; // 创建第二个结点
newnode->nextarc = NULL;
if (check[VR[i][1]] == 1) {
G.vertices[order[VR[i][1]]].firstarc = newnode;
check[VR[i][1]] = 0;
} else {
ArcNode *temp;
temp = G.vertices[order[VR[i][1]]].firstarc;
while (temp->nextarc)
temp = temp->nextarc;
temp->nextarc = newnode;
}
G.arcnum++;
}
}
}
return OK;
}
```

```
status DestroyGraph(ALGraph &G)
/* 销毁无向图 G, 删除 G 的全部顶点和边 */
{
for (int i = 0; i < G.vexnum; i++) {
```

```
ArcNode *temp = G.vertices[i].firstarc, *pre = temp;
while (temp) {
    temp = temp->nextarc;
    free(pre);
    pre = temp;
}
G.vertices[i].firstarc = NULL;
}
G.vexnum = 0;
G.arcnum = 0;
return OK;
}
```

```
int LocateVex(ALGraph G, KeyType u)
//根据 u 在图 G 中查找顶点，查找成功返回位序，否则返回-1;
{
    for (int i = 0; i < G.vexnum; i++)
        if (u == G.vertices[i].data.key)
            return i;
    return -1;
}
```

```
status PutVex(ALGraph &G, KeyType u, VertexType value)
//根据 u 在图 G 中查找顶点，查找成功将该顶点值修改成 value，返回 OK;
//如果查找失败或关键字不唯一，返回 ERROR
{
    int pos, flag = 0;
    for (pos = 0; pos < G.vexnum; pos++)
        if (u == G.vertices[pos].data.key) {
            flag = 1;
            break;
        }
}
```

```
}  
if (!flag)  
    return ERROR;  
for (int i = 0; i < G.vexnum; i++)  
    if (value.key == G.vertices[i].data.key)  
        return ERROR;  
G.vertices[pos].data = value;  
return OK;  
}
```

```
int FirstAdjVex(ALGraph G, KeyType u)  
//根据 u 在图 G 中查找顶点，查找成功返回顶点 u 的第一邻接顶点位序，否则返回-1;  
{  
    int firstvexorder = -1;  
    for (int i = 0; i < G.vexnum; i++)  
        if (u == G.vertices[i].data.key) {  
            firstvexorder = G.vertices[i].firstarc->adjvex;  
            break;  
        }  
    return firstvexorder;  
}
```

```
int NextAdjVex(ALGraph G, KeyType v, KeyType w)  
//根据 u 在图 G 中查找顶点，查找成功返回顶点 v 的邻接顶点相对于 w 的下一邻接顶点的位序，查找失败返回-1;  
{  
    int flag = 0, vo;  
    for (vo = 0; vo < G.vexnum; vo++)  
        if (v == G.vertices[vo].data.key) {  
            flag = 1;
```

```
break;
}
if (!flag)
return -1;
flag = 0;
for (int i = 0; i < G.vexnum; i++)
if (w == G.vertices[i].data.key) {
flag = 1;
break;
}
if (!flag)
return -1; // 分别判断 v、w 对应的顶点是否存在
ArcNode *temp = G.vertices[vo].firstarc;
while (temp->nextarc) {
if (G.vertices[temp->adjvex].data.key == w) {
temp = temp->nextarc;
for (int i = 0; i < G.vexnum; i++)
if (G.vertices[temp->adjvex].data.key == G.vertices[i].data.key) {
return i;
}
temp = temp->nextarc;
}
return -1;
}
}
```

```
status InsertVex(ALGraph &G, VertexType v)
//在图 G 中插入顶点 v，成功返回 OK，否则返回 ERROR
{
if (G.vexnum >= 20)
return ERROR;
```

```
int i;
for (i = 0; i < G.vexnum; i++)
if (v.key == G.vertices[i].data.key)
return ERROR;
G.vertices[i].data = v;
G.vertices[i].firstarc = NULL;
G.vexnum++;
return OK;
}
```

```
status DeleteVex(ALGraph &G, KeyType v)
```

//在图 G 中删除关键字 v 对应的顶点以及相关的弧，成功返回 OK, 否则返回 ERROR

```
{
int i, flag = 0;
ArcNode *pre, *next, *temp;
for (i = 0; i < G.vexnum; i++)
if (v == G.vertices[i].data.key) {
flag = 1; // i 是删除的结点位序
break;
}
if (!flag || G.vexnum <= 1) // 保证删除后图不空
return ERROR;
// 接下来释放该节点的链表空间
temp = G.vertices[i].firstarc, pre = temp;
while (temp) {
temp = temp->nextarc;
free(pre);
G.arcnum--;
pre = temp;
}
```

```
G.vertices[i].firstarc = NULL;

for (int j = i + 1; j < G.vexnum; j++)
G.vertices[j - 1] = G.vertices[j]; //数组全部前移
G.vexnum--;
for (int j = 0; j < G.vexnum; j++) {
if (!G.vertices[j].firstarc)
continue; // 如果不存在弧
temp = G.vertices[j].firstarc;
while (temp) {
if (temp->adjvex == i)
temp->adjvex = -999; //用-999 标记要删的弧结点
else if (temp->adjvex > i)
temp->adjvex--; //序号前移 1
temp = temp->nextarc;
}
pre = G.vertices[j].firstarc;
next = pre->nextarc; // 下面逐一删去位次为-999 的结点
if (!next) { // 只有一条弧的情况
if (pre->adjvex == -999) {
G.vertices[j].firstarc = NULL;
free(pre);
}
continue;
} else { // 有多条弧
//首先判断第一个点要不要删
if (pre->adjvex == -999) {
G.vertices[j].firstarc = next;
free(pre);
pre = next;
next = next->nextarc;
```



```
}  
//然后遍历后面所有点  
while (next) {  
    if (next->adjvex == -999) {  
        pre->nextarc = next->nextarc;  
        free(next);  
    }  
    next = next->nextarc;  
}  
}  
}  
return OK;  
}  
  
status InsertArc(ALGraph &G, KeyType v, KeyType w)  
//在图 G 中增加弧 <v,w>, 成功返回 OK, 否则返回 ERROR  
{  
    int vo, wo; // 分别是 v 和 w 的位序  
    for (vo = 0; vo < G.vexnum; vo++)  
        if (v == G.vertices[vo].data.key)  
            break;  
    if (vo == G.vexnum)  
        return ERROR;  
    for (wo = 0; wo < G.vexnum; wo++)  
        if (w == G.vertices[wo].data.key)  
            break;  
    if (wo == G.vexnum)  
        return ERROR;  
    ArcNode *temp = G.vertices[vo].firstarc;  
    while (temp) { //判断是否已存在路径 <v,w>  
        if (G.vertices[temp->adjvex].data.key == w)
```

```
return ERROR;
temp = temp->nextarc;
}

ArcNode *newnode = (ArcNode *)malloc(sizeof(ArcNode));
newnode->adjvex = wo;
newnode->nextarc = G.vertices[vo].firstarc;
G.vertices[vo].firstarc = newnode;
newnode = (ArcNode *)malloc(sizeof(ArcNode));
newnode->adjvex = vo;
newnode->nextarc = G.vertices[wo].firstarc;
G.vertices[wo].firstarc = newnode;
G.arcnum++;
return OK;
}

status DeleteArc(ALGraph &G, KeyType v, KeyType w)
//在图 G 中删除弧 <v,w>, 成功返回 OK, 否则返回 ERROR
{
int vo, wo, flag = 0;
for (vo = 0; vo < G.vexnum; vo++)
if (v == G.vertices[vo].data.key)
break;
if (vo == G.vexnum)
return ERROR;
for (wo = 0; wo < G.vexnum; wo++)
if (w == G.vertices[wo].data.key)
break;
if (wo == G.vexnum)
return ERROR;
```

```
ArcNode *temp = G.vertices[vo].firstarc, *pre, *next;
while (temp) { //判断路径 <v,w> 是否存在
if (G.vertices[temp->adjvex].data.key == w) {
flag = 1;
break;
}
temp = temp->nextarc;
}
if (!flag)
return ERROR;
```

```
pre = G.vertices[vo].firstarc;
next = pre->nextarc;
if (G.vertices[pre->adjvex].data.key == w) {
G.vertices[vo].firstarc = next;
free(pre);
} else {
while (next) {
if (G.vertices[next->adjvex].data.key == w) {
pre->nextarc = next->nextarc;
free(next);
break;
}
pre = pre->nextarc;
next = next->nextarc;
}
}
```

```
pre = G.vertices[wo].firstarc;
next = pre->nextarc;
if (G.vertices[pre->adjvex].data.key == v) {
```

```
G.vertices[wo].firstarc = next;
free(pre);
} else {
while (next) {
if (G.vertices[next->adjvex].data.key == v) {
pre->nextarc = next->nextarc;
free(next);
break;
}
pre = pre->nextarc;
next = next->nextarc;
}
}
G.arcnum--;
return OK;
}
```

```
void visit(VertexType v) {
printf("%d %s", v.key, v.others);
}
```

int visited[100];// 记录某点是否被访问过

```
status DFSTraverse(ALGraph &G, void (*visit)(VertexType))
```

//对图 G 进行深度优先搜索遍历，依次对图中的每一个顶点使用函数 visit 访问一次，且仅访问一次

```
{
if (G.vexnum < 1)
return ERROR;
ArcNode *temp;
for (int i = 0; i < G.vexnum; i++)
```

```
visited[i] = 0;
for (int i = 0; i < G.vexnum; i++)
if (!visited[i]) {
    visit(G.vertices[i].data);
    visited[i] = 1;
    temp = G.vertices[i].firstarc;
    while (temp) {
        if (!visited[temp->adjvex]) {
            visit(G.vertices[temp->adjvex].data);
            visited[temp->adjvex] = 1;
            temp = G.vertices[temp->adjvex].firstarc;
        } else
            break;
    }
}
return OK;
}
```

status BFSTraverse(ALGraph &G, void (\*visit)(VertexType))

//对图 G 进行广度优先搜索遍历，依次对图中的每一个顶点使用函数 visit 访问一次，且仅访问一次

```
{
if (G.vexnum < 1)
return ERROR;
ArcNode *temp;
int queue[100];// 存位序
int front = 0, rear = 0;
for (int i = 0; i < G.vexnum; i++)
    visited[i] = 0;
for (int i = 0; i < G.vexnum; i++)
    if (!visited[i]) {
```

```
queue[front] = i;
rear++;
temp = G.vertices[i].firstarc;
while (temp) {
    queue[rear++] = temp->adjvex;
    temp = temp->nextarc;
}
while (rear - front > 0) {
    if (!visited[queue[front]]) {
        visit(G.vertices[queue[front]].data);
        visited[queue[front]] = 1;
        temp = G.vertices[queue[front++]].firstarc;
        while (temp) {
            queue[rear++] = temp->adjvex;
            temp = temp->nextarc;
        }
    } else
        front++;
}
return OK;
}
```

```
status SaveGraph(ALGraph G, char FileName[])
//将图的数据写入到文件 FileName 中
{
    if (G.vexnum < 1)
        return INFEASIBLE;
    FILE *fp;
    fp = fopen(FileName, "w");
    ArcNode *temp;
```

```
int count;
fprintf(fp, "%d %d\n", G.vexnum, G.arcnum);
for (int i = 0; i < G.vexnum; i++) { // 存储每个顶点的邻接点数
temp = G.vertices[i].firstarc;
count = 0;
while (temp) {
count++;
temp = temp->nextarc;
}
fprintf(fp, "%d\n", count);
}
for (int i = 0; i < G.vexnum; i++) {
fprintf(fp, "%d %s", G.vertices[i].data.key, G.vertices[i].data.others);
temp = G.vertices[i].firstarc;
while (temp) {
fprintf(fp, " %d", temp->adjvex);
temp = temp->nextarc;
}
fprintf(fp, "\n");
}
fclose(fp);
return OK;
}

status LoadGraph(ALGraph &G, char FileName[])

//读入文件 FileName 的图数据，创建图的邻接表
{
if (G.vexnum)
return INFEASIBLE;
int count[30];
ArcNode *newnode, *temp;
```

```
FILE *fp;
fp = fopen(FileName, "r");
fscanf(fp, "%d %d\n", &G.vexnum, &G.arcnum);
for (int i = 0; i < G.vexnum; i++)
fscanf(fp, "%d\n", &count[i]);
for (int i = 0; i < G.vexnum; i++) {
fscanf(fp, "%d %s", &G.vertices[i].data.key, G.vertices[i].data.others);
if (count[i] == 0)
G.vertices[i].firstarc = NULL;
else {
newnode = (ArcNode *)malloc(sizeof(ArcNode));
fscanf(fp, " %d", &newnode->adjvex);
G.vertices[i].firstarc = newnode;
G.vertices[i].firstarc->nextarc = NULL;
temp = G.vertices[i].firstarc;
for (int j = 1; j < count[i]; j++) {
newnode = (ArcNode *)malloc(sizeof(ArcNode));
fscanf(fp, " %d", &newnode->adjvex);
temp->nextarc = newnode;
newnode->nextarc = NULL;
temp = newnode;
}
}
fscanf(fp, "\n");
}
fclose(fp);
return OK;
}
```

```
int Near(ALGraph &G, int vo, int wo)
```

```
// 判断两个顶点是否直接相连, 相邻返回 1, 否则返回 999999 (距离无限)
```



```
{
ArcNode *temp = G.vertices[vo].firstarc;
while (temp) {
if (temp->adjvex == wo)
return 1;
temp = temp->nextarc;
}
return INF;
}

int ShortestPathLength(ALGraph &G, KeyType v, KeyType w)
// 返回顶点 v 与顶点 w 的最短路径的长度
{
int i, j, flag, t, min, temp;
int pre[50], d[50];
int vo = LocateVex(G, v);
int wo = LocateVex(G, w);
if (vo == -1 || wo == -1)
return ERROR;
for (i = 0; i < G.vexnum; i++) {
visited[i] = 0;
pre[i] = 0;
d[i] = Near(G, vo, i);
}
visited[vo] = 1;
d[vo] = 0;
//下面每次循环都添加一条最短路径
for (i = 1; i < G.vexnum; i++) {
t = 0;
flag = -1;
min = INF;
```

```
for (j = 0; j < G.vexnum; j++)
if (!visited[j] && d[j] < min) { //寻找未添加的最短路径
min = d[j];
flag = j;
visited[flag] = 1;
}
if (flag == -1) //不连通
break;
for (j = 0; j < G.vexnum; j++) {
temp = Near(G, flag, j);
if (temp != INF)
temp += min;
if (!visited[j] && temp < d[j]) { //修改到 j 的最短路径
d[j] = temp;
pre[j] = flag;
t++;
}
}
}
if (d[wo] == INF)
return ERROR;
else
return d[wo];
}

void VerticesSetLessThanK(ALGraph &G, KeyType v, int k)
// 返回与顶点 v 距离小于 k 的顶点集合
{
int count = 0;
for (int i = 0; i < G.vexnum; i++)
if (G.vertices[i].data.key != v && ShortestPathLength(G, v, G.vertices[i].data.key) <
```

```
k) {  
    visit(G.vertices[i].data);  
    count++;  
}  
if (!count)  
    printf("不存在与该顶点距离小于%d 的顶点! \n", k);  
}
```

```
int ConnectedComponentsNums(ALGraph &G)
```

```
// 返回无向图连通分量的个数
```

```
{  
    if (G.vexnum < 1)  
        return ERROR;  
    int ConnectedComponents = 0;  
    ArcNode *temp;  
    int queue[100];  
    int front = 0, rear = 0;  
    for (int i = 0; i < G.vexnum; i++)  
        visited[i] = 0;  
    for (int i = 0; i < G.vexnum; i++)  
        if (!visited[i]) {  
            queue[front] = i;  
            rear++;  
            temp = G.vertices[i].firstarc;  
            while (temp) {  
                queue[rear++] = temp->adjvex;  
                temp = temp->nextarc;  
            }  
            while (rear - front > 0) {  
                if (!visited[queue[front]]) {  
                    visited[queue[front]] = 1;  
                    ConnectedComponents++;  
                    front++;  
                }  
            }  
        }  
    return ConnectedComponents;  
}
```

```
temp = G.vertices[queue[front++]].firstarc;
while (temp) {
    queue[rear++] = temp->adjvex;
    temp = temp->nextarc;
}
} else
    front++;
}
if (front == rear)
    ConnectedComponents++;
}
return ConnectedComponents;
}
```

```
int main() {
    int op = 1;
    int i, j, k, result, choice;
    char FileName[30], GraphName[30];
    ALGraph G;
    G.arcnum = 0;
    G.vexnum = 0;
    G.kind = UDG;
    VertexType V[30];
    VertexType value;
    KeyType VR[100][2];
    KeyType u, v, w;
    GRAPHS GGG;
    GGG.length = 0;
    while (op) {
        system("cls");
        printf("\n\n");
```

```
printf("-----\n");
printf(" 无向图操作演示系统 \n 计算机 2110 班李嘉鹏\n");
printf("-----\n");
printf(" 1. CreateGraph 2. DestroyGraph\n");
printf(" 3. LocateVex 4. PutVex\n");
printf(" 5. FirstAdjVex 6. NextAdjVex\n");
printf(" 7. InsertVex 8. DeleteVex\n");
printf(" 9. InsertArc 10. DeleteArc\n");
printf(" 11. DFSTraverse 12. BFSTraverse\n");
printf(" 13. Save/Load 14. VerticesSetLessThanK\n");
printf(" 15. ShortestPathLength 16. ConnectedComponentsNums\n");
printf(" 17. MultiGraph 0. Exit\n");
printf("-----\n");
printf(" 请选择操作 [0 17]:\n");
scanf("%d", &op);
switch (op) {
// 以下分别对应 17 个功能，根据输入情况执行对应的操作
case 1:
if (G.vexnum)
printf(" 无向图创建失败！ \n");
else {
printf(" 请输入顶点序列和关系对序列，分别以-1 nil 和-1 -1 结束： \n");
i = 0;
do {
scanf("%d%s", &V[i].key, V[i].others);
} while (V[i++].key != -1);
i = 0;
do {
scanf("%d%d", &VR[i][0], &VR[i][1]);
} while (VR[i++][0] != -1);
result = CreateCraph(G, V, VR);
```

```
if (result == OK)
printf(" 无向图创建成功! \n");
else if (result == ERROR)
printf(" 输入数据有误, 无向图创建失败! \n");
}
getchar();
getchar();
break;
case 2:
if (!G.vexnum)
printf(" 无向图为空! \n");
else if (DestroyGraph(G) == OK)
printf(" 无向图清空成功! \n");
getchar();
getchar();
break;
case 3:
if (!G.vexnum)
printf(" 无向图为空! \n");
else {
printf(" 请输入要查找顶点的关键字: ");
scanf("%d", &u);
result = LocateVex(G, u);
if (result == -1)
printf(" 查找失败! \n");
else
printf(" 关键字为%d 的结点在图中的位序为%d! \n", u, result);
}
getchar();
getchar();
break;
```

```
case 4:
if (!G.vexnum)
printf(" 无向图为空! \n");
else {
printf(" 请输入要赋值顶点的关键字、新的关键字和名称，用空格隔开: ");
scanf("%d %d %s", &u, &value.key, value.others);
result = PutVex(G, u, value);
if (result == OK)
printf(" 赋值成功! \n");
else if (result == ERROR)
printf(" 查找失败或关键字不唯一! \n");
}
getchar();
getchar();
break;
case 5:
if (!G.vexnum)
printf(" 无向图为空! \n");
else {
printf(" 请输入要查找第一邻接点的顶点关键字: ");
scanf("%d", &u);
result = FirstAdjVex(G, u);
if (result == OK)
printf(" 关键字为%d 的顶点的第一邻接点位序为%d! \n", u, result);
else if (result == -1)
printf(" 查找失败或该顶点不存在邻接点! \n");
}
getchar();
getchar();
break;
case 6:
```

```
if (!G.vexnum)
printf(" 无向图为空! \n");
else {
printf(" 请输入要查找的顶点关键字和其某一邻接点的关键字，用空格隔开： ");
scanf("%d %d", &v, &w);
result = NextAdjVex(G, v, w);
if (result == OK)
printf(" 关键字为%d 的顶点的邻接点%d 的下一邻接点的位序为%d! \n", v, w);
else if (result == -1)
printf(" 查找失败! \n");
}
getchar();
getchar();
break;
case 7:
if (!G.vexnum)
printf(" 无向图为空! \n");
else {
printf(" 请输入新顶点的关键字和名称，用空格隔开： ");
scanf("%d %s", &value.key, value.others);
result = InsertVex(G, value);
if (result == OK)
printf(" 顶点%d %s 插入成功! \n", value.key, value.others);
else if (result == ERROR)
printf(" 插入失败! \n");
}
getchar();
getchar();
break;
case 8:
if (!G.vexnum)
```



```
printf("无向图为空! \n");
else {
printf("请输入要删除顶点的关键字: ");
scanf("%d", &v);
result = DeleteVex(G, v);
if (result == OK)
printf("关键字%d 对应的顶点与相关的弧删除成功! \n", v);
else if (result == ERROR)
printf("删除失败! \n");
}
getchar();
getchar();
break;
case 9:
if (!G.vexnum)
printf("无向图为空! \n");
else {
printf("请输入要增加的弧两端顶点的关键字, 用空格隔开: ");
scanf("%d %d", &v, &w);
result = InsertArc(G, v, w);
if (result == OK)
printf("已成功增加顶点%d 和%d 之间的弧! \n", v, w);
else if (result == ERROR)
printf("弧增加失败! \n");
}
getchar();
getchar();
break;
case 10:
if (!G.vexnum)
printf("无向图为空! \n");
```

```
else {
printf(" 请输入要删除的弧两端顶点的关键字，用空格隔开：");
scanf("%d %d", &v, &w);
result = DeleteArc(G, v, w);
if (result == OK)
printf(" 已成功删除顶点%d 和%d 之间的弧！ \n", v, w);
else if (result == ERROR)
printf(" 弧删除失败！ \n");
}
getchar();
getchar();
break;
case 11:
if (!G.vexnum)
printf(" 无向图为空！ \n");
else {
printf("\n—————深度优先搜索遍历结果—————\n\n");
DFS_Traverse(G, visit);
printf("\n—————end—————\n");
}
getchar();
getchar();
break;
case 12:
if (!G.vexnum)
printf(" 无向图为空！ \n");
else {
printf("\n—————广度优先搜索遍历结果—————\n\n");
BFS_Traverse(G, visit);
printf("\n—————end—————\n");
}
```

```
getchar();
getchar();
break;
case 13:
printf(" 请输入目标文件名: ");
scanf("%s", FileName);
if (G.vexnum) {
result = SaveGraph(G, FileName);
if (result == OK)
printf(" 无向图数据已成功写入文件%s! \n", FileName);
else
printf(" 写文件失败! \n");
} else {
result = LoadGraph(G, FileName);
if (result == OK)
printf(" 文件%s 中的数据已成功读入到无向图中! \n", FileName);
else
printf(" 文件写入失败! \n");
}
getchar();
getchar();
break;
case 14:
if (!G.vexnum)
printf(" 无向图为空! \n");
else {
printf(" 请输入顶点的关键字和 k 值 (距离小于 k), 用空格隔开: ");
scanf("%d %d", &v, &k);
printf("\n————与顶点%d 距离小于%d 的顶点集合————\n\n", v, k);
VerticesSetLessThanK(G, v, k);
printf("\n\n————end————\n\n");
```

```
}
getchar();
getchar();
break;
case 15:
if (!G.vexnum)
printf(" 无向图为空! \n");
else {
printf(" 请输入两个顶点的关键字，用空格隔开: ");
scanf("%d %d", &v, &w);
result = ShortestPathLength(G, v, w);
if (result == ERROR)
printf(" 输入数据错或这两个顶点之间不存在通路! \n");
else
printf(" 顶点%d 和%d 之间最短路径的长度为%d! \n", v, w, result);
}
getchar();
getchar();
break;
case 16:
if (!G.vexnum)
printf(" 无向图为空! \n");
else {
result = ConnectedComponentsNums(G);
if (result == ERROR)
printf(" 无向图中无顶点! \n");
else
printf(" 无向图中连通分量的个数为%d! \n", result);
}
getchar();
getchar();
```

```
break;
case 17:
printf(" 请选择功能： 1. 无向图添加； 2. 无向图移除； 3. 无向图查找\n");
scanf("%d", &choice);
if (choice == 1) {
printf(" 请输入新无向图名称： ");
scanf("%s", GGG.elem[GGG.length].name);
GGG.elem[GGG.length].G.arcnum = 0;
GGG.elem[GGG.length].G.vexnum = 0;
printf(" 请输入顶点序列和关系对序列，分别以-1 nil 和-1 -1 结束： \n");
i = 0;
do {
scanf("%d%s", &V[i].key, V[i].others);
} while (V[i++].key != -1);
i = 0;
do {
scanf("%d%d", &VR[i][0], &VR[i][1]);
} while (VR[i++][0] != -1);
result = CreateCraph(GGG.elem[GGG.length].G, V, VR);
if (result == OK) {
GGG.length++;
printf(" 无向图%s 添加成功！ \n", GGG.elem[GGG.length - 1].name);
} else if (result == ERROR)
printf(" 输入数据有误，无向图创建失败！ \n");
} else if (choice == 2) {
if (GGG.length <= 0)
printf(" 删除失败，无向图集合为空！ \n");
else {
printf(" 请输入无向图名称： ");
scanf("%s", GraphName);
int flag = 0;
```

```
for (j = 0; j < GGG.length; j++)
if (strcmp(GGG.elem[j].name, GraphName) == 0) {
flag = 1;
break;
}
if (flag) {
for (int k = j; k < GGG.length - 1; k++) {
GGG.elem[k].G = GGG.elem[k + 1].G;
strcpy(GGG.elem[k].name, GGG.elem[k + 1].name);
}
GGG.length--;
printf(" 无向图%s 已从无向图集合中删除! ", GraphName);
} else
printf(" 无向图集合中没有名为%s 的无向图! \n", GraphName);
}
} else if (choice == 3) {
if (GGG.length <= 0)
printf(" 无向图集合为空! \n");
else {
printf(" 请输入无向图名称: ");
scanf("%s", GraphName);
int flag = 0;
for (i = 0; i < GGG.length; i++)
if (strcmp(GGG.elem[i].name, GraphName) == 0) {
flag = 1;
break;
}
if (flag) {
G = GGG.elem[i].G;
printf(" 查找成功! \n");
} else
```

```
printf("无向图集合中没有名为%s的无向图!\n", GraphName);
}
} else
printf("请重新选择!\n");
getchar();
getchar();
break;
case 0:
break;
}
}
printf("欢迎下次再使用本系统!\n");
return 0;
}
```